# Crafting Adversarial Examples using Particle Swarm Optimization

Rayan Mosli
rhm6501@rit.edu

# Crafting Adversarial Examples using Particle Swarm Optimization

by

Rayan Mosli

A dissertation submitted in partial fulfillment of the
requirements for the degree of
**Doctor of Philosophy**
**in Computing and Information Sciences**

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
April 24th 2020

# Crafting Adversarial Examples using Particle Swarm Optimization

by

Rayan Mosli

**Committee Approval:**

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

---

Dr. Yin Pan — Date
Dissertation Advisor

---

Dr. Xumin Liu — Date
Dissertation Committee Member

---

Dr. Bo Yuan — Date
Dissertation Committee Member

---

Dr. Rui Li — Date
Dissertation Committee Member

---

Dr. Thomas Smith — Date
Dissertation Defense Chairperson

**Certified by:**

---

Dr. Pengcheng Shi — Date
Ph.D. Program Director, Computing and Information Sciences

# Crafting Adversarial Examples using Particle Swarm Optimization

by

Rayan Mosli

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences
Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
**Doctor of Philosophy Degree**
at the Rochester Institute of Technology

## Abstract

Machine learning models have been found to be vulnerable to adversarial attacks that apply small perturbations to input samples to get them misclassified. Attacks that search for and apply the perturbations are performed in both white-box and black-box settings, depending on the information available to the attacker about the target. For black-box attacks, the attacker can only query the target with specially crafted inputs and observing the outputs returned by the model. These outputs are used to guide the perturbations and create adversarial examples that are then misclassified.

Current black-box attacks on API-based malware classifiers rely solely on feature insertion when applying perturbations. This restriction is set in place to ensure that no changes are introduced to the malware's originally intended functionality. Additionally, the API calls being inserted in the malware are null or no-op APIs that have no functional affect to avoid any unintentional impact on malware behavior. Due to the nature of these API calls, they can be easily detected through non-ML techniques by analyzing their arguments and return values.

In this dissertation, we explore other attacks on API-based malware detection models that are not restricted to feature addition. Specifically, we explore feature replacement as a possible avenue for creating adversarial malware examples. To retain the malware's original functionality, we replace API calls

with other functionally equivalent API calls. We find the API alternatives by using a hierarchical unsupervised learning approach on the API's documentation. Our attack, which we call AdversarialPSO, uses Particle Swarm Optimization to guide the perturbations according to available function alternatives. Results show that creating adversarial malware examples by feature replacement is possible even under the more restrictive search space of limited function alternatives.

Unlike the malware domain, which lacks benchmark datasets and publicly available classification models, image classification has multiple benchmarks to test new attacks. Therefore, to evaluate the efficacy and wide-applicability of AdversarialPSO, we re-implement the attack in the image classification domain, where we create adversarial examples from images by adding small often unrecognizable perturbations to the inputs. As a result of these perturbations, highly-accurate models misclassify the inputs resulting in a drastic drop in their accuracy. We evaluate this attack against both defended and undefended models and show that AdversarialPSO performs comparably to state-of-the-art adversarial attacks.

## Acknowledgments

There are many who stood by me and supported me during my PhD journey and to them all I owe a great debt of gratitude. I would first like to thank my advisor Dr. Yin Pan, whose knowledge and expertise were invaluable to my research. She was patient with me as I made mistake after mistake and she always made sure that I learned from each and every one of them. My research would not have been possible without her. I am truly blessed to have had her as my advisor.

Dr. Bo Yuan taught me my first malware course, and ever since, he has never stopped being my teacher and mentor. He gave me opportunities that allowed me to grow both within and outside my research interests. He trusted me with responsibilities in both teaching and advising, and just as he had a hand in laying my foundations in malware, he also helped shape my career as an educator. For all of that, I want to express my most sincere gratitude to Dr. Yuan.

I would also like to thank Dr. Rui Li for guiding me through the oceans of machine learning. Although I still have much to learn, a lot of what I know can be traced back to him. Dr. Li always had his door open for me and for that I will always be extremely grateful.

The direction of my research was largely influenced by simple questions asked by Dr. Xumin Liu. Without her knowledge, the outcome of this dissertation would have been vastly different. I would like to express my deepest thanks to Dr. Liu for her encouragement, insightful comments, and hard questions.

Dr. Matthew Wright's guidance during my time working on this dissertation was simply indispensable. His feedback always pushed me forward and his comments were always meant to elevate my work. I am immensely thankful for all his help.

There are also many that worked behind the scenes to ensure my success. From our director Dr. Pengcheng Shi, to the administrative and technical operations staff, Min-Hong Fu, Lorrie Jo Turner, Amanda Zeluff, Charles Gruener, and Dave Emlen. Thank you all for making my time in the program seem like a breeze.

I would also like to acknowledge my fellow lab-mates in The Center for Cybersecurity and my fellow PhD students in the GCCIS PhD program. During my time at RIT, I had the pleasure of knowing many extraordinary individuals, many of whom became my close friends. Thank you all for making my

time here at RIT and in Rochester among my best memories.

I would not have reached this far without the support of my parents Faten and Hisham, and my siblings, Hala, Mahmoud, Mohammed, Yasmin, and Rana. Their continuous and unconditional love was the anchor I needed to weather the storm.

Last but not least, I would like to thank my wife Shahd for being my rock. Her unwavering support was crucial to my emotional and mental well-being. Because of her, I never had to worry about anything other than my research. Although I am sure she will always remind me of that, I hope this acknowledgement will assure her that I will never forget.

*For Shahd, Elyas, Faten, and Hisham, you are my compass for whenever I get lost.*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

With the proliferation of electronic devices and the ever-increasing dependency on technology, malware has become an attractive tool for malicious activities. As reported by Kaspersky Lab [39], approximately 717 million malware attacks were detected by the anti-virus (AV) tool in the second quarter of 2019 alone. The increase in malware sightings worldwide is largely due to their profitability, where malware authors realize gains through black-market sales of malware, selling stolen data, or from ransom payments demanded after ransomware attacks. This increase in malicious software activity threatens the online safety of both organizations and individuals alike.

Machine learning (ML) has been extensively explored in the malware literature as a possible avenue for better and more reliable detection. In fact, it has already been incorporated in many commercial anti-malware products [40,49,77]. This adoption is attributed to the highly accurate and generalizable predictions offered by ML. The utility of ML has been demonstrated in many different faucets of malware analysis and detection, such as the classification of malware to their families (e.g., Zbot, Koobface, Banker, and Virut) [26], quantifying the similarity between different malware samples [14], and detecting variants spawned from the same malware roots [81].

Despite state-of-the-art performances, ML models have been shown to suffer from a general flaw that makes them vulnerable to external attack. Adversaries can manipulate models to misclassify inputs by applying small perturbations to the samples [78]. These *adversarial examples* have been successfully demonstrated in the image classification domain against real-world black-box

targets, where adversaries would perform remote queries on a classifier to develop and verify their attack samples [60]. These attackers were later adopted in the malware field to create adversarial malware examples that evades ML detection [13,18,24,68]. The possibility of such attacks poses a significant risk to any ML application, especially in security-critical settings or life-threatening environments.

Early adversarial attacks relied on model gradients to craft adversarial examples [11, 17, 61], which requires internal knowledge of the target model. Such information is available in a white-box attack but not in a black-box setting. This led some black-box attacks to rely on the ability of some adversarial examples to transfer from one model to another [60]. These attacks train a local surrogate that approximates the target's decision boundary to craft adversarial examples. Adversarial examples crafted on the local surrogate are then transferred to the remote target where they would also be misclassified. Transfer-based attacks however, were shown to be ineffective [12], which led to the exploration of other types of attacks that do not require a surrogate. These methods attack the model directly by either estimating the model gradients [6,12,29,30] or by iteratively applying perturbations to the input, guided by certain algorithms [2,21,52].

In practice, the feasibility of a black-box attack also depends heavily on the number of required queries submitted to the model. Against machine-learning-as-a-service (MLaaS) platforms like Google Vision, each query has a monetary cost. Too many queries make the attack costly. Perhaps more importantly, too many queries could trigger a monitor to detect an attack underway by observing many subtly modified versions of the same input submitted to the system in a short period. To evade such a monitor, one could conduct the attack very slowly or use a large number of accounts that all have different credit cards attached and different IP addresses. Either approach would significantly add to the real-world costs of conducting the attack.

Malware and images are inherently different and their attacks are bound by different constraints. The main constraint when generating adversarial examples from images is the distance from the original input, which must not be too large that the semantic meaning of the image is lost. In other words, changes made to the image to create the adversarial example must not distort the image so that objects in the image are no longer recognizable. For malware on the other hand, the constraint is to create adversarial examples

that maintain the malware's original functionality. Put differently, changes made to the malware must not change the output of running the malware on a victim machine.

To maintain this constraint when creating adversarial malware examples, researchers limit perturbations to feature additions, which ensures that malware functionality would remain intact [13,18,24,68]. Attacks for both static-based models [13, 18] and for dynamic-based models [24, 68] have generally avoided feature replacement and removal to maintain the functionality of the malware. However, limiting perturbations to feature additions introduces a weakness that makes the attack detectable. As the features being inserted must maintain the malware's functionality, they consist exclusively of no-op or null features that have no effect. As such, these features can be detected by scanning the binary for extraneous artifacts or by analyzing the behavioral artifacts that are produced when running the malware in a sandbox [64].

In the case of dynamic ML-based malware detection models, specifically those that use API calls, samples are first executed in a sandbox and monitored before they are classified by an ML model. Analyzing the arguments being passed to the API calls and the return values they produce, could help determine which API calls are no-ops that do not contribute to the software's intended goals [64]. For example, if an API call returns a value that is not used later during malware execution, then that API call serves no purpose and can be safely removed. Another more specific example is if a file is opened using OpenFile (returns a file handle) but is not followed by a ReadFile or a WriteFile (both require a file handle as a parameter), then that API call can be safely removed. As a consequence, attacks that depend on inserting null API calls into the malware call sequence are easily detectable using non-ML techniques.

This weakness in API insertion attacks led us to explore a more robust approach for creating adversarial examples. Specifically, we aim to answer the following Research Question (RQ):

- **RQ1:** How can adversarial examples for API-based malware detection models be created without relying solely on null feature insertion?

To answer **RQ1**, we pursue a method that is capable of replacing API calls made by the malware with functionally equivalent API calls. We design an attack, which we call AdversarialPSO, that is capable of generating adversarial

examples from malware without being limited to feature additions. This attack would ensure that all API calls made by the malware would be relevant, and detecting such calls during dynamic analysis would be much more difficult than detecting null or no-op API calls. Chapter 5, describes our findings for answering **RQ1**.

AdversarialPSO uses Particle Swarm Optimization (PSO)—a gradient-free optimization technique—to craft adversarial examples. PSO maintains a population of candidate solutions called particles. Each particle moves in the search space seeking better solutions to the problem based on a fitness function that we have designed for finding adversarial examples. PSO has been shown to quickly converge on good (though not globally optimal) solutions [74], making it very suitable for finding adversarial examples in a black-box setting, as it can identify sufficiently good examples with few queries.

One frequently cited challenge when performing research in the malware domain is the lack of benchmark datasets and publicly available models. This however, is not a issue in the image classification domain. Many adversarial attacks are first proposed for image classification models and are later adapted to malware detection models. This led us to wonder about how well our attack would fare against state-of-the-art attacks in image classification. Although the two domains are inherently different, which we discuss further in Section 4.2, it would nonetheless be insightful to evaluate the PSO-based attack under different constraints. This would help determine which aspects of our attack requires further attention, thus guiding our future work in this area. We therefore pursue a second research question **RQ2**, which is as following:

- **RQ2:** Can the AdversarialPSO attack, which is designed for API-based malware detection models, be effectively adapted for image classification models?

In answering **RQ2**, we determined that the attack algorithm itself performs comparably to state-of-the-art attacks in image classification, and any effort to improve the malware attack would be better spent in finding or implementing more functionally-equivalent API calls. We report our findings for **RQ2** in Chapter 6.

## Contributions and Summary

To better understand why ML is actively being explored in current malware research, in Chapter 2, we provide a brief background on traditional approaches for malware analysis and detection. We also disuss why traditional approaches are no longer sufficient in the current malware landscape. Furthermore, we discuss how ML is used to overcome the shortcomings of traditional malware detection approaches and we provide a brief introduction on adversarial ML and the different adversarial threat models.

Essentially, the primary focus of this dissertation is to explore robust attacks against commonly used malware detection models. For this evaluation, we trained a ML model on API call sequences collected from both malware and benignware- an approach often used in the malware literature. Chapter 3 provides further details on the malware detection model we developed for this study.

The attack we designed, called AdversarialPSO, is a black-box attack that creates adversarial examples for malware detection models. We use the population-based evolutionary search algorithm Particle Swarm Optimization as a basis for our attack. The AdversarialPSO attack generates adversarial examples by making targeted changes to a malware's API call sequence using the PSO algorithm. Also, as the original functionality of the malware must be retained, the attack replaces API calls made by the malware with functionally equivalent API calls. AdversarialPSO, including our process for finding functionally equivalent APIs, are discussed in more detail in Chapter 5.

As the malware domain lacks both benchmark datasets and publicly available models, it is thus difficult to evaluate the efficacy of the attack on reliable baselines. Therefore, we also test the attack in the image classification domain, which contains several benchmarks. This allows us to compare the effectiveness and practicality of the attack under different constraints. One example of how the two domains are different is that, unlike malware, which are represented by discrete features, images reside in a continuous feature-space. This requires adapting our original AdversarialPSO attack, which was initially designed to operate on discrete features, to work on images. The results of using this attack on image classification models is presented in Chapter 6.

The main contributions of this study are as following:

- We design and evaluate a feature replacement attack against API-based

malware detection models

- We explore methods for finding functionally equivalent APIs that could be used in feature replacement attacks
- We evaluate the attack in the image classification domain and show that our attack performs comparably to the state-of-the-art in image classification adversarial ML attacks

# Chapter 2

# Background & Literature Review

In this chapter, we first provide some background on traditional malware detection approaches, how ML is used to detect malware, and some introductory concepts in the field of adversarial ML. We then review the literature that is related to our work.

## 2.1 Background

### 2.1.1 Traditional Approaches for Malware Analysis and Detection

To detect malware, we currently rely on AV software to scan and clean our systems. The vast majority of those AVs use approaches that can be easily evaded by malware. Namely, they use signatures from previously seen malware, often in the form byte sequences and hash digests, to detect future sightings of the same malware. Although these approaches are indispensable for the detection of many known malware variants, they fail when encountering recently developed malware with unknown signatures, commonly known as zero-days. Additionally, malware authors often use obfuscation techniques, such as polymorphism, metamorphism, and packing, to modify malware signatures and evade AV detection. With these vulnerabilities, traditional signature-based approaches for detecting malware has become insufficient on their own and

therefore must be supplemented with other detection methods.

An alternative to using signatures for malware detection is the use of behavior, that of both malware and benign software. Using behavior to detect malware can be done by either looking for deviancy from normal system behavior, or by searching for similarities to known malicious behavior. The former, known as anomaly-based detection, is done by setting a threshold for normal system activities and raising an alarm whenever that threshold is crossed. The latter on the other hand, involves executing the malware in isolated environments and monitoring their behavior. The observed behavior is used to detect software that show similarities to known malware activities. Although better in detecting unknown and obfuscated malware than signature-based detection, behavioral-based approaches have weaknesses that affect their broad applicability. Examples of those weaknesses include high false positive rates, moving thresholds due to new applications and usage patterns, the extensive resources required to execute and observe the malware, and the inability to directly invoke all malware execution paths. The weaknesses of both signature-based and behavioral-based approaches prompted researchers to explore other avenues for better malware detection.

Researchers saw potential in ML to provide solutions that overcome the weaknesses of signature-based and behavioral-based malware detection. ML has been already proven in other fields as an indispensable tool that provides highly accurate predictions, and based on that, researcher explored its adoption in the malware domain. Studies that use ML in the malware field are often categorized according to the objective of the study, the features used, and the ML techniques being utilized [80]. The most common objective is the detection of malware, which aims to classify processes as either malicious or benign. For the remainder of this dissertation, we consider this objective as our primary focus for malware classification. We will discuss the different types of features in the next section, but we refer the readers to the survey by Ucci et al. [80] for a more detailed discussion on the different ML techniques used in the malware field.

### 2.1.2 Machine Learning in the Malware Domain

There are two types of features when training ML models for malware detecting: static-based and behavioral-based features. Static features are extracted

directly from software binaries without executing the samples, such as operational code (opcode) sequences [72], Control Flow Graphs(CFG) [3], header information [47], and byte entropy [73]. The challenge when using static features is handling the obfuscation, packing, and encryption that is often used by malware developers. Without addressing these evasive measures, the extraction of static features will fail to provide meaningful intelligence to the learning algorithms. Although there are methods to overcome these challenges, they are rarely universal and would often require some manual work, which hinders the scalability and portability of the models.

Behavioral-based features on the other hand, requires running the malware in sandboxes to extract the necessary features. Examples of behavioral features include API calls [22], network activity [56], memory usage [67], and file system activity [51]. This approach is vulnerable to anti-sandbox technology [50], which if not addressed, could pollute the dataset with mislabeled data points. For samples that use anti-sandbox or environment-aware technology, executing the malware in a sandbox would fail to invoke their original behavior. In those instances, if the malware detects any trace of an analysis environment, they would simply terminate or limit their behavior to benign activities. Behavioral-based features also suffer from incomplete coverage of malware code. When malware is executed, only a single execution path is taken in each run, whereas many possible paths could still exist in the code. Without a complete picture of all potential behaviors within the malware, the ML models would be trained with a narrow view of the feature-space.

There are avenues that combine both static and behavioral features, and are less vulnerable to obfuscation. One such avenue is memory, which represents the runtime state of the computer system in a single point in time. Any software that executes must pass through memory in its native form before executing. In other words, encrypted software must decrypt and packed executables must unpack. This offers an opportunity to extract both static and behavioral features while overcoming some of the challenges that are often encountered when analyzing malware. Features could also be extracted from memory images instead of a live system, this would ensure that no live malware is tampering with the analysis process. The applications of this however, might be limited as memory analysis and artifact extraction could be time-consuming and resource-intensive. In Section 2.2.1, we discuss our previous work in memory-based malware detection and the benefits it may offer to the

field of computer forensics.

### 2.1.3 Adversarial Machine Learning

Although ML could effectively detect malware in non-hostile settings, its robustness could still be questioned when the models come under attack. Recently, the research community has shown great interest in the area of adversarial ML [5, 17, 75, 79]. In short, the goal of this field is to subvert ML models and cause them to misclassify inputs. The mere possibility of these kind of attacks can have a large impact on the applicability of ML in any domain, especially malware detection. If malware developers can evade detection by making small targeted changes to their malware, this would place any ML-based malware detection model under direct threat of subversion.

Adversarial ML attacks are categorized using three dimensions: influence (specifies the attacker's capabilities), security violation (integrity, availability, or privacy), and specificity (the breadth of the attack) [58]. The combination of the aforementioned dimensions creates the threat model of the attack. Specifying the threat model is essential when conducting research in the adversarial ML domain, both from the attacker's perspective and that of the defender's.

The first dimension in an adversarial threat model is the attacker's capabilities. An attacker can have influence on the training data and launch an attack before the model is trained. These attacks are known as poisoning attacks [58], as they aim to influence the model's decision by poisoning the training set. Without influence on the training data, the attacker is left with exploratory capabilities, which limits the attacker's control to manipulating the test samples and querying an already trained model for feedback. Attacks of this type are often launched in a black-box setting where the attacker has no knowledge of the target [60]. It is important to consider the number of queries submitted to the model when launching exploratory attacks. If an attack submits too many queries, it might raise suspicion. It is therefore essential for these type of attacks to use limited queries for them to be practical.

Another dimension to consider when categorizing adversarial attacks is the security policy being violated by the attacker. When attacking the availability of the model, the attacker aims to render the model completely useless by producing misclassifications on all or most test samples. On the other hand,

attacks on integrity only aims to misclassify specific samples, which consequently, will reduce the overall confidence of the model. Finally, by attacking the privacy of the model, an attacker attempts to leak information from the training data by exploiting design flaws in the model itself.

The third and final dimension is the breadth or specificity of the attack. This dimension specifies the amount of samples targeted by the attacker, where the goal is to either misclassify specific samples or to indiscriminately target all samples. This dimension is closely related to the security policy dimension, where specific and indiscriminate attacks would indirectly lead to attacks on integrity and availability, respectively. Furthermore, specificity and influence are also related, as depending on the attacker's capabilities, different attacks would directly influence either specific samples or all samples. For example, if the attacker launches a poisoning attack that influences the model's decision boundary, any sample classified by the model would be affected by this change, which would make it an indiscriminate attack. On the other hand, if the attacker can only perturb certain features in the input sample, then the changes would only affect that sample.

Adversarial ML has been studied in the domains of image recognition [60], speech and voice recognition [1], and also in security [13, 18, 83]. ML models in each of these areas were successfully subverted using adversarial ML, which bolsters the need for developing robust models that withstands external attacks. Although there are multiple defenses proposed in the literature, only one defense was found to be effective, and that is adversarial training. This defense involves generating attack samples and incorporating them in the training set to make the model less sensitive to adversarial perturbations [27, 46].

## 2.2 Literature Review

To clearly present related work from the literature, we separate the review into the following three sections: malware detection, adversarial ML, and adversarial ML in the malware domain.

### 2.2.1 Malware Detection

The malware detection ML literature can be divided according to the type of features used to train the ML models. Malware can be represented using

two types of features: static features (e.g., binary n-grams, opcode sequences, and metadata), and dynamic features (e.g., DNS requests, accessed files, and registry activity). The main difference between the two is the method used to capture the data- static features are collected without executing the malware while dynamic features requires executing the malware and monitoring their behavior. ML models can be trained using one of the types exclusively or by combining the two to provide the model with a more complete view of the feature space. Hybrid approaches that use both static and dynamic features often perform better, but they incur a higher overhead for feature extraction and selection. On the other hand, having multiple feature sets offer the opportunity to train a Multi-Classifier System (MCS) that would be more robust than a single classifier. Although the work in this dissertation focuses on dynamic-based malware detection, we include static-based approaches for the sake of completeness.

**Malware Detection using Static Features**

A compiled program is represented by a sequence of opcodes and operands that are executed by the CPU. In essence, opcodes are the instructions that make up the program's operations and the operands are the parameters used by those opcodes. In a study by Bilar, a significant statistical difference in opcode usage was found between benignware and malware [7]. Based on this, Santos et al. used Term Frequency (TF) to extract features from opcode sequences and Mutual Information (MI) to select the most relevant features [72]. Using the frequencies and relevance scores, the authors built a Support Vector Machine (SVM) classifier based on a normalized polynomial kernel, which achieved an accuracy of 95.9%.

More et al. explore ensemble voting techniques for malware detection [53]. The features they use consisted of opcode data extracted using n-grams, overlapping n-grams, and sliding window. They reduce the number of features using Term Frequency-Inverse Document Frequency (TF-IDF) by selecting the $n$ most highest scoring features. The authors then test multiple voting schemes including majority voting, veto voting, and trust-based veto voting in their ensemble methods for malware detection. They found trust-based veto voting to achieve the best performance on their dataset, with the highest accuracy being 89.7% on overlapping n-gram features.

To classify binaries as benign or malicious, the authors of [66] used raw byte sequences extracted directly from binary files. Over two million time steps were used to represent each binary, which imposed unique challenges to the authors. Once the byte sequences were extracted, the authors used Convolution Neural Networks (CNN) with global max pooling to classify binaries as malicious or benign. The purpose of the study was not to achieve high-detection rates, but to explore the challenges of using long sequences and to identify sub-regions in the binary that would not have been identified otherwise.

**Malware Detection using Dynamic Features**

Although using static features has its advantages, it is still vulnerable to packed and encrypted samples. As pointed out in [57] and as was demonstrated in [36], packing, encryption, k-ary code, and multistage loaders are challenges to static-based ML approaches. For that reason, many researchers choose dynamic approaches for malware classification and detection. Although dynamic approaches also have weaknesses, they are considered to be better in generalizing to unknown malware and in the prediction of future trends.

Pirscoveanu et al. used DNS requests, accessed files, mutexes, modified registry keys, and API calls to train a Random Forest (RF) classifier for malware detection [65]. They acquired the data using Cuckoo Sandbox while utilizing INetSim to simulate internet connectivity. They achieved 96% accuracy with a weighted average of 89.9% across all malware types.

AMAL, proposed by Mohaisen et al., is a behavioral-based malware classification system that characterizes samples according to filesystem, registry, network, and memory artifacts [51]. The system consists of two subsystems, one to run the samples and extract the features, called AutoMal, and the other to build the classifier, called MaLabel. According to the authors, AMAL achieved an accuracy of 99.5%. One of the strengths of AMAL is the large training set used to build the model, where the authors used more than 115,000 samples to train their models.

Huang et al. performed multi-task learning on two objectives functions, one for the binary classification of malware and the other for classifying malware to their respective families [28]. They utilized API call sequences and null terminated objects recovered from system memory using anti-malware

engines. Both objective functions were trained simultaneously using the same architecture, which splits into two softmax layers for producing both binary and multi-class labels. They report error rates of 0.35% for binary classifications and 2.94% for multi-class classification.

To predict the maliciousness of executables, Rhode et al. trained an ensemble of Recurrent Neural Networks (RNN) on machine activity data, such as total number of running processes, CPU and memory usage, and network communications [67]. The objective of this study was to detect malware in the first four seconds of execution. The authors report a 93% accuracy for their model, however, they also mention an obvious counter-measure of their approach, which is to inject benign behavior in the first few second of malware execution.

Kolosnjaji et al. combine recurrent and convolutional layers to train a deep learning model for malware detection [37]. They use API call sequences to train DL models that classifies sequences to specific malware families. They found that combining convolutional and recurrent improves the classification of malware behavior.

### Other Avenues for Malware Detection

One avenue that could benefit greatly from the utility of ML is digital forensics. Recently, memory forensics has assumed a larger role in the digital forensics investigative process. This process involves acquiring memory images from a live system and analyzing them for evidence of maliciousness or malpractice. Part of that process is to search for traces of malware on a computer system, as in many computer crimes, malware is often involved one way or another. Although there are many tools to assist in the analysis of memory images, it is often a manual process that largely depends on the analysts' expertise. This leads to a time consuming investigation that is both resource intensive and error-prone. To solve this problem, we propose using behavioral artifacts found in memory images to classify processes as malicious or benign. We explore memory artifacts such as registry keys, referenced APIs, loaded libraries, and open handles as possible features for malware detection [54, 55].

In our analysis, we found great disparity between malware and benignware usage of memory artifacts. For example, we found that malicious processes, on average, use more than twice as many process handles as bengin processes.

This is a result of malware attempting to take over other process for stealth and privilege escalation purposes. Another example is the usage of certain API calls that are highly indicative of anti-analysis intentions. Specifically, we found that it is more common for malware to import APIs such as Sleep and GetTickCount than legitimate software. These APIs are often called to evade automated analysis or to detect the presence of debuggers, respectively. Finally, we found registry keys related to network tracing and remote access services to be accessed more frequently by malware than benignware. The common occurrence of such registry keys in the malware corpus could be a result of data exfiltration operations and the preparation of victim machines for remote access by the malware developers.

After extracting and analyzing the memory artifacts, we trained ML models to classify processes in a memory image as malicious or benign. We test both individual classifiers for each type of artifact and an ensemble-based classifier that aggregates individual classifications. The ensemble achieved an accuracy of 93.39% and precision and recall of 90.58% and 96.25%, respectively. Our results show that the detection of malware during the course of a memory investigation can be automated using ML.

### 2.2.2 Adversarial Machine Learning

#### White-box Attacks

Adversarial examples were first discussed by Szegedy et al. in [79] where they were described as an exploitation of blind spots in a neural network's coverage of the feature space. These blind spots cause neural networks to become sensitive to changes in the input, where small perturbations lead to large variations in the output. Utilizing this sensitivity, multiple algorithms were proposed to fool classifiers into misclassifying inputs.

Goodfellow et al. argue that the linearity of neural networks is the cause of adversarial examples. Based on this, they introduced the Fast Gradient Sign Method (FGSM) [17]. This method, shown in 2.1, makes small changes to the inputs towards cost function's gradient $\nabla_x J(\theta, \boldsymbol{X}, y)$, which can be computed using back-propagation.

$$\eta = \epsilon sign(\nabla_{\boldsymbol{X}} J(\theta, \boldsymbol{X}, y)) \tag{2.1}$$

Where $\eta$ is the perturbation to be added to the input sample $\boldsymbol{X}$ and $\epsilon$ is the

magnitude of the perturbation. The authors show that a single step towards the gradient is often sufficient to get a sample misclassified.

Kurakin et al. [38] extend this approach by introducing an iterative variant of FGSM that takes several smaller steps instead of a single large step. The authors also evaluate the persistence of both FGSM and its iterative variant in the physical world. To do so, they print out adversarial examples and classify the images by feeding them to a model through a camera. They found that adversarial attacks are possible even in the physical world.

Another approach is the Jacobian-based Saliency Map Approach (JSMA) proposed by Papernot et al. [62]. In contrast to FGSM, which makes small changes to a large number of features, JSMA makes large changes to a small number of features. The method, shown in 2.2, determines which features to modify by creating an adversarial saliency map generated from the Jacobian of the network. JSMA can be used in a targeted attack where perturbations can be made to misclassify a sample to a specific class. Both FGSM and JSMA were used in [60] to launch a black-box attack on a remote model. The attack depends on the transferability phenomena discussed in [61], where adversarial examples generated for one model would transfer to another model, even if the two models have different architectures or were trained using different learning algorithms. This allowed the authors of [60] to train a local surrogate classifier that approximates the decision boundary of the target. The surrogate was then used to generate adversarial examples using JSMA and FGSM, which would then transfer to the target model and be misclassified.

$$S(\boldsymbol{X}, t)[i] = \begin{cases} 0 \, if \dfrac{\partial \boldsymbol{F}_t(\boldsymbol{X})}{\partial \boldsymbol{X}_i} < 0 \, or \sum_{j \neq t} \dfrac{\partial \boldsymbol{F}_j(\boldsymbol{X})}{\partial \boldsymbol{X}_i} > 0 \\ \left( \dfrac{\partial \boldsymbol{F}_t(\boldsymbol{X})}{\partial \boldsymbol{X}_i} \right) \left| \sum_{j \neq t} \dfrac{\partial \boldsymbol{F}_j(\boldsymbol{X})}{\partial \boldsymbol{X}_i} \right| \, otherwise \end{cases} \tag{2.2}$$

Where $S(\boldsymbol{X}, t)[i]$ is the saliency map created to misclassify input sample $\boldsymbol{X}$ as target class label $t$ and where $i$ is an input feature.

At the time of writing, the state-of-the-art in white-box adversarial attacks was the Carlini & Wagner attack (C&W) [11], in which the authors search for adversarial examples by iteratively performing
$minimize \, \mathcal{D}(x, x + \delta)$, where $\mathcal{D}$ is either an $L_0$, $L_2$, or $L_\infty$ distance metric. The attack finds the minimum distance required to generate an adversarial example according to the distance metric being minimized. The attack defeats the defensive distillation approach proposed by Papernot et al. [62].

### Black-box Attacks

In a black-box attack, the attacker does not know the internals of a target model. Instead, the attacker can query the target with specially crafted inputs. Target models are assumed to return confidence scores along with each classification, which is then used in constructing inputs for subsequent queries. These inputs are crafted to estimate gradient or to lead to generating misclassification samples gradually.

**Gradient-Estimation Attacks:** To launch black-box attacks on image classification models, Chen et al. propose ZOO [12], a method to estimate model gradients using only the model inputs and the corresponding confidence scores provided by the model. The approach employs a finite difference method that evaluates image coordinates after adding a small perturbation to estimate the direction of the gradient for each coordinate. However, as examining every coordinate does require a large number of evaluations, the authors applied the stochastic coordinate descent algorithm and attack-space dimension reduction to reduce the number of evaluations needed to approximate gradients. Small perturbations added to the direction of the gradient, which, as shown in the FGSM attack, are sufficient to obtain an adversarial example from the input. Although it can successfully create adversarial examples indistinguishable from the inputs, the ZOO attack requires up to a million queries for high-dimensional data such as Imagenet. With so many queries, the attack could be easily detectable, and the cost could be prohibitive and impractical in a real-world setting for a single image.

To mitigate the large number of queries required by ZOO, Bhagoji et al. estimate the gradient of groups of features or coordinates instead of estimating one coordinate at a time [6]. Although the attack was not evaluated on a high-dimensional dataset, it outperformed ZOO on low-dimensional datasets such as CIFAR-10 and MNIST. The proposed Gradient Estimation (GE) approach used by the authors still requires up to 10 thousand queries to generate an adversarial example. The authors considered PSO as a possible approach for searching adversarial examples but found it to be slow and not as useful as GE. As we show in Chapter 6, however, our modifications to the basic PSO algorithm enable it to outperform GE. Our version of PSO does not require a swarm of 100 particles to be effective, which would be slow as per Bhagoji

et al.'s experience. Consequently, it can search for adversarial examples with high success rates using swarms with as few as five (5) particles against image classification models.

Ilyas et al. propose Natural Evolutionary Strategies (NES) to estimate column-wise gradients to find adversarial examples [29]. The authors use projected gradient descent on the estimated gradients to craft adversarial examples. They also extend the approach in [30] to utilize the bandit optimization method to exploit prior information when estimating the gradients. Specifically, they incorporate a data-dependent prior, which exploits the similarity in gradient information exhibited by adjacent pixels. Furthermore, they also incorporate a time-dependent prior that utilizes the high correlation between gradients estimated in successive steps. Although the attack can generate high-quality adversarial examples with few queries, the approach has been shown to be quite sensitive to changes in hyperparameter values. Moon et al. [52] have shown that having too many hyperparameters could lead to significant variability in attack performance, creating dependability on the values chosen for those hyperparameters. Gradient-estimation based approaches commonly have multiple hyperparameters that are necessary for the execution of attacks such as the learning rate, search variance, decay rate, and update rules – in a real-world black-box setting, tuning these hyperparameters would either incur additional queries or might not be possible at all in many cases. In our black-box attack on image classification models, there are only two hyperparameters and they have predictable effects on the outcome of the attack.

**Gradient-Free Attacks:** Moon et al. formulate the problem of crafting adversarial examples as a set maximization problem that searches for the set of positive and negative perturbations that maximizes an objective function [52]. Similar to [30], the authors exploit the spatial regularity exhibited by adjacent pixels by searching for perturbations in blocks instead of individual pixels. They increase the granularity of the blocks as the search progresses. Our AdversarialPSO attack searches for perturbations in blocks as well and yields comparable results as Moon et al.'s approach. However, our approach is capable of adjusting hyperparameter values effectively for the trade-off between L2 and queries as we show in Chapter 6.

Guo et al. explore a simple attack that crafts adversarial examples by randomly sampling a set of orthonormal vectors and adding or subtracting them

from the input [21]. The attack is shown to be successful in crafting adversarial examples despite its simplicity. However, the success of the attack diminishes as dimensionality increases, as shown when targeting InceptionV3, which expects inputs (299x299) with higher dimensionality than that of ResNet and DenseNet (224x224). As the perturbations are applied randomly, many queries are wasted by the approach until a solution is found.

By utilizing Differential Evolution (DE), Su et al. show that some test samples can be misclassified by changing a single pixel [76]. Similar to the PSO algorithm used in this paper, DE is a population-based algorithm that maintains and manipulates a set of candidate solutions until an acceptable outcome is found. The objective of this one-pixel attack is to better understand the geometry of adversarial space and proximity of adversarial examples to their corresponding inputs. The attack does not achieve high success rates due to the tight constraints used in the study.

Another population-based black-box attack is GenAttack [2], which uses Genetic Algorithm (GA) to find adversarial examples. This attack iteratively performs the three genetic functions *selection*, *crossover*, and *mutation*, where selection extracts the fittest candidates in a population, crossover produces a child from two parents, and mutation encodes diversity to the population by applying small random perturbations. The authors propose two heuristics to reduce the number of queries used by GenAttack, namely dimensionality reduction and adaptive parameter scaling. Although the authors propose two heuristics to reduce the numbers of queries used by their approach, GenAttack uses a higher number of queries compared to our approach.

### Defenses

Multiple defenses against adversarial examples were proposed, which focuses on either making the architecture more robust, detecting adversarial examples, or including the adversarial examples in the training process. Papernot et al. used distillation to make the model more robust to perturbations [63]. Distillation is traditionally used to transfer knowledge from a larger network to a second smaller network. However, the authors of [63] used distillation to reduce the number of network gradients that can be targeted by attackers, thus making the model more robust to changes.

Another defense was proposed by Gu et al., where they increase model

robustness by using Deep Contractive Networks (DCN) [19]. In essence, DCN generalizes Contractive Autoencoders (CAE) to a feedforward neural network. The purpose of DCNs is to minimize output variance with respect to the input by incorporating layer-wise penalties and thus making the network more robust.

To detect and defend against adversarial examples, Meng et al. introduced MagNet [48]. MagNet uses two networks: a detector to detect adversarial examples and a reformer to reshape them back to the manifold of normal samples. This approach does not make modifications to the original architecture but complements it with MagNet to detect and protect against adversarial examples. To detect attacks, the authors use the reconstruction error of autoencoders, where the sample is considered adversarial if the reconstruction error was high. Furthermore, if the reconstruction error was low, they would utilize the probability divergence between the reconstruction error and the target classifier's softmax output to determine whether or not the sample is adversarial.

To fortify their model against adversarial examples in the malware domain, the authors of [27] generated attack samples and incorporated them in their training process. They tested four different methods to generate adversarial examples, which are then included in the training process using the saddle point formulation presented in [46]. The malware features used by the authors to evaluate this approach were statically extracted API calls from PE binaries.

### 2.2.3 Adversarial ML in the Malware Domain

Grosse et al. [18] utilized the saliency map attack introduced in [62] to generate adversarial examples for the Android malware detection model DREBIN [4]. However, as JSMA was designed for the continuous feature-space of images, the authors had to adapt the attack to the discrete feature-space of malware. Similar to the original JSMA implementation, the authors use the model's gradient to determine which features would have the largest impact of the model's prediction. However, in contrast to applying JSMA on images, which would add perturbations to pixel values, the attack on malware would simply set the binary values representing the malware features to indicate the presence of specific API calls that were originally absent from the malware. In other words, the attack was restricted to adding features to the malware and not

removing or replacing features. This restriction was enforced to maintain the malware's original functionality as removing features would have a more drastic impact of how the malware behaves than adding features.

Another approach for generating adversarial malware examples was proposed by Demontis et al. [13], where the authors also generate adversarial examples to mislead DREBIN [4]. Five different attack scenarios were addressed in the study based on the attacker's capabilities and knowledge of the target. Specifically, the authors address a no-effort attack that performs no transformations on the malware, a DexGaurd-based attack that uses the Android obfuscation tool DexGaurd to transform malware, a mimicry attack in which the attacker trains a surrogate classifier that approximates the target, a limited-knowledge attack in which the attacker knows the learning algorithm used by the target, and finally, a perfect-knowledge attack, which assumes the attacker has internal knowledge of the target classifier. For the mimicry, limited-knowledge, and perfect knowledge attacks, the authors utilize an optimization formula that maximizes the probability of evasion by modifying features in the malware. The purpose of these scenarios is to evaluate a defense proposed by the authors.

To generate adversarial malware examples in a black-box setting, Hu and Tan use Generative Adversarial Networks (GANs) in their attack called MalGAN [24]. MalGAN uses a substitute model that approximates the target and a generative network that minimizes malware predictions of the substitute model. The model targeted in this paper uses API calls found in the code to make predictions and similar to [18], the authors restrict modifications to adding features to the malware. Adversarial examples are created by passing both the feature-vector representing the malware and a randomly-generated noise vector to the generator. The output of the generator indicates which absent API calls to inject in the malware code to produce misclassifications by the substitute model. The attack depends on the transferability of adversarial examples from the substitute model to the target black-box.

Similar to [24], Rosenberg et al. also use a surrogate Recurrent Neural Network (RNN) and a GAN to generate adversarial examples from malicious API call sequences [68]. Furthermore, their attack also depends on the trasferability property of adversarial examples. The authors use the GAN to generate benign no-op API calls which they then inject into the malware's API call sequence. No-op API calls are API calls that have no functional effect on the

execution flow of the malware. Almost any API can be made into a no-op API by passing certain arguments when the API is called. For example, passing an invalid file handle to the API call OpenFile to open a non-existent file. Another type of no-op API calls are those that merely queries for information without modifying the sytsem, such as the GetSystemDirectoryA API call, which returns the path of the system directory. The issue however, with inserting such APIs in the malware call sequence is that they are easily detectable. The malware can be scanned for APIs calls with invalid arguments and unused return values. For that reason, we choose to pursue an attack that replaces API calls with functionally equivalent API calls to create a more powerful and robust attack.

# Chapter 3

# Malware Detection using API Call Sequences

In this chapter, we discuss the details of training the malware detection model we use as a target for the AdversarialPSO attack. The model is trained on API call sequences made by both malware and benignware. At the time of writing this dissertation, there were no publicly available API call datasets for training ML models. Therefore, we generate our own dataset by executing the samples in a sandbox environments and recording the API calls made by each sample. We chose the dynamic-analysis route for data collection and malware detection for two reasons. First, we chose dynamic-analysis to avoid the complexity and possible bias introduced by static-analysis evasion techniques commonly used by malware developers. Although dynamic-analysis evasion is still possible, they are often easier to recognize during the analysis process if they were employed by a malware sample. We discuss later in the chapter our steps to mitigate the effects of such evasion techniques. The vulnerability of static-analysis to evasive measures has led many researchers to utilize dynamic approaches for analyzing malware. The increased usage of dynamic-analysis approaches in the literature is the second reason we chose this route for evaluating our replacement-based adversarial attack. Nonetheless, as we discuss in Section 5.2.3, attacking static-based models using the same approach is possible. Attacking static-based classifiers using our replacement attack is an avenue we would like to explore in future work. In the following sections, we discuss our process for data collection, pre-processing, and model training.

## 3.1 Data Collection

To elevate the quality of malware experiments, Rossow et al. proposed 19 guidelines to handle, describe, and analyze malware datasets [69]. These guidelines are grouped into four categories: correctness, transparency, realism, and safety. Each of these categories addresses a specific requirement essential to the success of research projects in the malware domain. Namely, the correctness category addresses the general health of a malware dataset to alleviate bias towards certain types of software. For example, one such guideline is to remove benignware from the malware portion of the dataset to avoid bias towards benign artifacts when training malware detection models. This could also occur indirectly if malware samples limit their behavior to benign activities due to the detection of an analysis environment, thus introducing benign behavior into the malware dataset. The transparency category contain guidelines that increase the repeatability and comprehensibility of the experiments. An example of a transparency guideline is to include a description of the analysis environment when reporting the results of a research project. Guidelines in the realism category are intended to increase the applicability of the proposed detection models in real-world scenarios by creating realistic environments and by evaluating the experiments under real-world conditions. For example, as malware commonly check for user interaction before executing, appropriate stimuli must be used in the experiments to ensure the proper execution of the malware. The last of the four categories is safety, which is concerned with the legal and ethical issues when conducting malware experiments, and the possible harm that may be inflicted on others. We attempt to follow these best practices when creating the dataset and analyzing the samples.

In the dynamic-analysis malware literature, API call sequences are among the most common artifacts used for malware detection [28, 35, 65]. To obtain these artifacts, a sandbox environment is required to execute both malware and benignware. Our sandbox was operated using Cuckoo Sandbox [1] and consisted of two Windows 7 SP1 virtual machines hosted on a Ubuntu 16.04 system. The Ubuntu system itself was a virtual machine hosted on a Windows 10 bare-metal system. The Ubuntu VM served three purposes: 1) it acted as a barrier between the Windows 7 VMs and the Windows 10 host, protecting

[1] `https://www.cuckoosandbox.org/`

it from live malware; 2) it ran Cuckoo sandbox, which automates the behavior analysis of software and records the API calls made by each sample; and 3) it simulated an internet connection to the Windows 7 VMs to increase the chance for successful malware execution. The internet was simulated using INetSim [2], which responds with fake websites, executables, and files when a sample requests any such resource, and it would also respond to many other different services such as DNS requests. The two Windows 7 VMs were run concurrently, each with a different sample and were allowed to run for four minutes for each sample. Benign software requiring installation were allowed to complete their installation process and then run for two additional minutes. Each of the Windows 7 VMs was configured with 4 GB of RAM and 1 CPU core. The Ubuntu system, on the other hand, had 16 GBs of RAM and 4 CPU cores. For each executed sample, Cuckoo produces a report of the sample's behavior containing the API call sequence made by the sample. Furthermore, using VirusTotal [3], we confirm the ground truth of each sample and the families of the malware samples.

Within the analysis VMs, we install a collection of commonly used software such as PDF readers, media players, browsers, text editors, spreadsheet programs, and email clients. This is to ensure that if a malware attempts to run a file with a certain extension, it would be able to do so through one of these programs. We also disable User Account Control (UAC), which would prompt the user before making any changes to the OS including running new software. Furthermore, we disable the firewall to allow the Cuckoo server to communicate freely with the analysis machines. Finally, we disable Windows Defender to avoid malware being blocked and quarantined by the default Windows anti-malware system.

We obtained the malware samples for this study from VirusShare [4]. We initially analyze 5000 malware samples containing Trojans, Worms, Viruses, Backdoors, and Adwares. Some samples were not classified to a category due to a disagreement between VirusTotal AVs, we retain them nonetheless as they were still classified as malicious. Benign software were collected from clean Windows 7 installations and software websites such as FileHippo [5]. We

[2] `https://www.inetsim.org/`
[3] `https://www.virustotal.com/`
[4] `https://www.virusshare.com/`
[5] http://www.filehippo.com/

obtained a total number of 3773 benign software samples from all software categories (e.g., browsers, file sharing, compression, multimedia, and security). After executing the malware in the sandbox, several samples terminated prematurely due to anti-VM techniques or unmet environment conditions, such as missing DLL files that are required for execution. We discard these samples from the malware training set, leaving us with 3837 malware samples. The total dataset size of both malware and benignware is 7610 samples, which we split into a training set (80%), validation set (10%), and test set (10%).

For each sample submitted for analysis, Cuckoo Sandbox starts a preconfigured VM, injects it with a sample, executes the sample, monitors its behavior, stops the VM, and reverts it back to a clean state. Cuckoo monitors the behavior of each sample by injecting a monitor into the initial process and in all its child processes [6]. The monitor places user-mode hooks that redirects execution to a hook handler whenever APIs are called. The APIs are then logged by the handler, which then returns execution to the hooked function.

## 3.2 Data Preprocessing

The next step is to preprocess the data for model training. The longest sequences in our dataset has over 1 million time-steps while the average sequence being approximately 25 thousand time-steps. However, upon inspecting the longer sequences, we found that many of the samples were repeatedly calling the same APIs, creating long sequence with too much redundancy. Although the repetition of API calls might have had a functional purpose for the sample (i.e., APIs called in a loop until a condition is met), retaining them makes the sequences too long, which would have an adverse effect on both training time and accuracy. This is shown by Raff et al. [66], where they explore the training of static-based malware detection models on sequences that contain up 2 million time-steps. In their study, they find that long sequences pose unique challenges, including the amount of resources needed to train the model. Therefore, we retain only three consecutive calls of the same API. In doing so, the average length of API call sequences went down from 25 thousand to 5000 time-steps and the longest sequence became 100 thousand time-steps.

After trimming the sequences to manageable lengths, we encode the API

[6] `https://www.cuckoo-monitor.readthedocs.io/en/latest/index.html`

calls to a format usable by the learning algorithm. We use label encoding to transform the text-based API calls to integers that represent their locations in the vocabulary. Although one-hot encoding in generally recommended when encoding categorical data, we tested both approaches and found label encoding to perform better. As we will explain in Section 3.3, we use an embedding layer in our deep learning architecture, which learns during the training process an appropriate transformation from discrete data to continuous data. Using the embedding layer with label encoding eliminated the need for one-hot encoding.

In addition to the evaluation of different encoding schemes, we also evaluate the optimal sequence length for training the model. We empirically evaluate different sequence lengths by training the model using variable sized inputs and testing on the validation set. We evaluate lengths of 500, 1000, 2500, 5000, and 10000 time-steps to find that the model accuracy plateaus after 2500 time-steps. Thus, we choose to train the model on the first 2500 API calls made by each sample. We discuss training the model in more detail in the next section.

## 3.3 Model Training

As shown by Kolosnjaji et al. [37], combining convolutional and recurrent layers improves the classification of API call sequences. Therefore, we combine the two layers to create a 5-layer DL model for malware classification. Our model consists of an embedding layer, 2 convolutional layers, a recurrent layer, and a fully connected layer followed by a softmax output layer to provide the class probabilities. We train the model using the binary cross-entropy loss and an Adam optimizer. We also use dropout in between the two convolutional layers to prevent overfitting. The full model can be seen in Figure 3.1.

For hyperparameter tuning, we use 5-fold cross validation. We tune the input sequence length, the embedding vector length, the kernel size and number of filters for the convolution layers, the pool size for max pooling, and the number of LSTM units in the recurrent layer. Table 3.1 shows the hyperparameter values we tested and the values we eventually use for training the malware detection model.

The model achieved an accuracy of 88.20% on the test set with 91.45% recall and 86.25% precision. The process of creating the malware detection model can be seen in Figure 3.2.

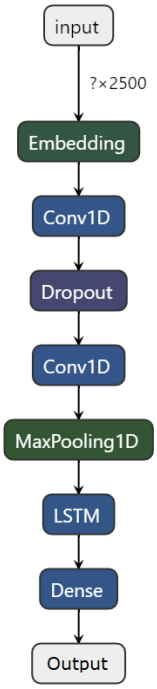


Figure 3.1: Malware Detection Deep Learning Model Architecture

| Hyperparameter | Values Tested | Best Value |
|---|---|---|
| Input Sequence Length | [500,1000,2500,5000,10000] | 2500 |
| Embedding Vector Length | [10,100,300,1000,5000] | 300 |
| Kernel Size | [2,3,4,5,6] | 6 |
| Pool Size | [2,3,4,5,6] | 5 |
| Number of LSTM Units | [10,50,100,200] | 100 |

Table 3.1: Hyperparameter Tuning Results

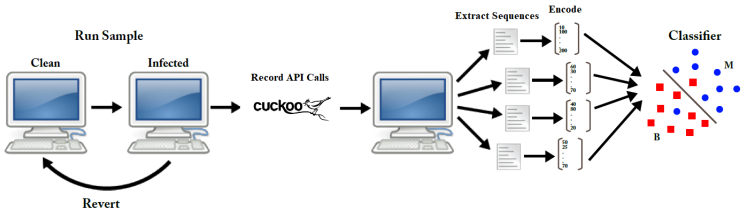


Figure 3.2: Overall Process for Training the Malware Detection Machine Learning Model

# Chapter 4

# Conventional Particle Swarm Optimization

To generate adversarial examples from both malware and images, we utilize the population-based algorithm Particle Swarm Optimization (PSO). In this chapter, we cover the fundamentals of PSO, which will be the foundation of the AdversarialPSO attacks that will be discussed in later chapters. Also, as the original PSO algorithm was designed to operate on continuous data, and as malware is often represented by discrete data, we discuss the necessary adaptations required to execute the attack on discrete data. Finally, to avoid redundancy in later chapters, we include some common aspects shared between the malware attack and the image classification attack.

## 4.1 Particle Swarm Optimization

Kennedy and Eberhart first proposed PSO as a model to simulate how flocks of birds forage for food [33]. It has since been adapted to address a multitude of problems, such as text feature selection [45], grid job scheduling [31], and optimizing the generation of electricity [16]. The algorithm works by dispersing particles in a search space and moving them until a solution is found. The search space is assumed to be $d$-dimensional, where the position of each particle $i$ is a $d$-dimensional vector $X_i = (x_{i,1}, x_{i,2}, x_{i,3}, \ldots, x_{i,d})$.

The position of each particle is updated according to a velocity vector $V_i$

where $V_i = (v_{i,1}, v_{i,2}, v_{i,3}, \ldots, v_{i,d})$. In each time-step or iteration, denoted as $t$, the velocity vector is used to update the particle's next position, calculated as:

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{4.1}$$

$$v_i(t+1) = wv_i(t) + c_1R_1(p_g - x_i(t)) + c_2R_2(p_i - x_i(t)) \tag{4.2}$$

Equation 4.2 contains three terms. The first term controls how much influence the current velocity has when calculating the next velocity and is constrained with the *inertia* weight $w$. The second term, with weight $c_1$, is referred to as *exploration*, as it allows particles to explore further regions in the search space in the direction of the best position found by the swarm, denoted by $p_g$. The third term, with weight $c_2$, is referred to as *exploitation*, and it is based on the best position found by this particle, denoted by $p_i$.

$R_1$ and $R_2$ are $d$-dimensional vectors that contain uniformly distributed random numbers which are calculated for each iteration to encode randomness in the search process.

Early implementations of PSO assigned a fixed value to $w$. Shi and Eberhart, however, found that linearly decreasing *inertia* weight improved PSO performance [74]. In each iteration, fixed values $w_{\text{start}}$ and $w_{\text{end}}$ together with a maximum number of iterations $t_{\max}$ were used to calculate $w$ as following:

$$w(t) = w_{\text{end}} + (w_{\text{start}} - w_{\text{end}}) \left( \frac{t_{\max} - t}{t_{\max}} \right) \tag{4.3}$$

## 4.2 Continuous vs Discrete Targets

To compute the velocity for each particle, the PSO algorithm calculates the distance between the particle's current position $X_i$, and both its best individual position $p_i$ and the best swarm position $p_g$. For continuous data, calculating these distances can be done directly using Equation 4.2. However, for discrete data, Equation 4.2 cannot be used directly to move particles and must be adapted to accommodate discrete search spaces.

The simplest way to adapt PSO to discrete and combinatoral problems is to use PSO normally as if on continuous data and then round off the values to the nearest valid point after each iteration [41]. However, as discussed in [32], this approach has two drawbacks that make it less effective than other discrete PSO approaches. First, by rounding continuous values to the nearest discrete

point, the destination of particle movements could lie outside the boundaries of the search space thus providing infeasible solutions. Furthermore, large discrepancies in the fitness function could be observed before and after rounding off the continuous numbers causing sub-optimal particle movements.

Another approach for solving discrete problems, called Binary PSO (BPSO), is proposed by Kennedy and Eberhart [34]. This approach requires the position vectors to be represented by binary values where every dimension in each candidate solution should contain either 0 or 1. The position vectors are then used to calculate the velocity vector using Equation 4.2, which is then transformed to a [0,1] interval using the sigmoid function. The outcome is a probability vector where each index is the probability of the same index in the position vector to take a value of 1, and where a low probability favors a value of 0. An extension of this approach was proposed by Veeramachaneni et al. to account for multi-valued discrete variables [82].

Pampara et al. proposed transforming high dimensional discrete search spaces to smaller continuous search spaces using an angle modulation-based method [59]. Similar to [34], this approach assumes binary values for each element in the position vector. The benefit of the angle modulated approach is that no modifications are required to the original PSO algorithm as the search space transformation is sufficient to accommodate discrete values.

The approach we use for attacking the malware detection model is a combination of the binary PSO approach proposed by Kennedy et al. [34] and its multi-valued variant proposed by Veeramachaneni et al. [82]. We discuss our attack further in Chapter 5. We also refer the readers to [32] for more information about using PSO in a discrete domain.

## 4.3 AdversarialPSO Foundations

Among the many applications of PSO, we show in this dissertation that it can also be used to craft adversarial examples for both images and malware. Shi and Eberhart [74] found that PSO is quick to converge on a solution and scales well to large dimensions, at the cost of slower convergence to global optima. This would make PSO an excellent fit for finding adversarial examples in the black-box setting, as it suggests that it can identify sufficiently good examples with few queries.

In the following section, we describe the commonalities between our attack

on malware detection models and our attack on image classification models.

### 4.3.1 Fitness Function

To adapt PSO to the problem of creating adversarial examples, we define a fitness function that measures the change in model output when perturbations are added to the input. In both targeted and untargeted attacks, the fitness function measures how much the model's confidence in the target label rises or drops, respectively. When performing untargeted attacks, the fitness for each candidate solution is the confidence drop in the original class predicted by the model. Given the original image $x$, the perturbed image $x'$, the model parameters $\theta$, and the original label $y$ we compute confidence $f(x, y, \theta)$. We then calculate the fitness using fitness $= f(x, y, \theta) - f(x', y, \theta)$. In targeted attacks, however, fitness is given by the *increase* in confidence in the desired class. For the target label $y'$, we compute confidence $f(x, y', \theta)$ and fitness $= f(x', y', \theta) - f(x, y', \theta)$.

### 4.3.2 Calculating the Inertia Weight $w$

As shown in Equation 4.3, the inertia weight $w$ is traditionally computed using the current and maximum number of iterations. However, in the case of black-box adversarial attacks, number of queries is a more appropriate measure for how much the attack progressed. Therefore, we modify Equation 4.3 to compute $w$ with respect to the number of queries instead of number of iterations as following:

$$w(t) = w_{\text{end}} + (w_{\text{start}} - w_{\text{end}}) \left( \frac{q_{\max} - q}{q_{\max}} \right) \tag{4.4}$$

where $q_{max}$ is the query budget used in the attack and $q$ is the number of queries submitted to the model. We use 1 and 0 for $w_{\text{start}}$ and $w_{\text{end}}$, respectively.

### 4.3.3 The Particle Explosion Problem

For long running attacks, the velocity would eventually become so large that it would overpower the exploration and exploitation terms in Equation 4.2. This would cause particles to get stuck at the edges of the search space as the ever-increasing velocity would continuously push them to locations outside the

valid boundaries of the search area. This is a well known problem in PSO and although the inertia weight is meant to mitigate it, it does not completely solve the problem. Therefore, in addition to the inertia weight, we perform velocity clamping to limit the growth of the velocity vector. This is performed at every iteration for each particle before applying the perturbation to the particle positions, and is performed as follows:

$$v_i(t) = clip(v_i(t), -B, B) \tag{4.5}$$

Where $-B$ and $B$ are the lower and upper bounds of the search space and are specific to the model under attack.

### 4.3.4 PSO Algorithm

The overall algorithm for using PSO to generate adversarial examples is the same for both malware detection and image classification. However, there are some differences at the implementation level, which we will discuss further in later chapters. For both attacks, the search for adversarial examples is performed in two stages: *initialization* and *optimization*. The *initialization* stage disperses the particles in the search space and tests the initial fitness for the starting point of each particle. The *optimization* stage moves the particles according to Equation 4.2 and tests the fitness for each new position until either an adversarial example is found or the query budget is exhausted, whichever comes first.

#### Initialization

For each input, the search process starts with initializing the particles by randomizing their positions in the search space. Particles are initialized by randomly perturbing an equal number of elements for each particle. In large swarms, each particle is assigned fewer elements, resulting in a more fine-grained search for adversarial examples. The overall algorithm for the initialization stage can be seen in Algorithm 1. A more detailed description of how the particles are initialized by each attack will be provided in each attack's respective chapters.

**Algorithm 1** Initializing the swarm

```
1: Input: input x, particle array par
2: bestFitness ← 0 # swarm-wide best
3: bestPosition ← x
4: divide search space among particles
5: for p in par do
6:     p.position ← x
7:     p.bestFitness ← bestFitness
8:     p.bestPosition ← bestposition
9:     perturb p.position
10:    fitness ← calculateFitness #includes update to q
11:    compare fitness against best particle fitness
12:    compare :fitness against best swarm fitness
13: end for
14: return par, bestPosition, bestFitness
```

### Optimization

The optimization step of AdversarialPSO (Algorithm 2) is an iterative process that moves the particles in search of better fitness. Particle positions are updated using the velocity vector, which is calculated for each particle in every iteration. After moving the particles, their fitness is calculated and compared against the particle's best fitness to determine which particle position will be used to calculate future particle movements. The particle's fitness is also compared against the best fitness achieved in the swarm as a whole (i.e, best swarm fitness), and if the particle fitness was found to be better, the swarm is updated to account for the position with the highest fitness. The process is repeated until an adversarial example is found or when the process exhausts the allowed number of queries.

Finally, we have adopted the mutation concept from genetic algorithms to encode more randomness in particle movements [15]. Randomly mutating particles helps avoid getting stuck in local minimas as there is always the possibility of moving the particles. This however, is implemented differently for both attacks so we leave their explanations to later chapters.

**Algorithm 2** Move Particles

1: **Input:** particle array $par$, swarm-wide best fitness $bestFitness$, and swarm-wide best position $bestPosition$
2: **for** $p$ *in* $par$ **do**
3: $v \leftarrow calculateVelocity$
4: $p.position \leftarrow updatePosition$
5: $fitness \leftarrow calculateFitness$ #includes update to $q$
6: $p.currentFit \leftarrow fitness$
7: compare fitness against best particle fitness
8: compare :fitness against best swarm fitness
9: **end for**
10: **return** $bestPosition$

# Chapter 5

# Adversarial Attack on Malware Detection Models using PSO

In this chapter, we describe our PSO attack on API-based malware detection models. Attacking models trained on software behavior requires that changes made to the input do not disrupt the original function of the software. In other words, the perturbations that fool the ML models must not introduce any errors to the software code. To accommodate this constraint, related work restrict their changes to adding features to the input [25,68]. In the case of API calls, the inserted APIs must be no-ops, which are APIs that have no effect on the execution flow of the software. Almost all APIs can be made into no-ops by controlling their arguments, for example, opening a non-existing file by passing an incorrect file handle. However, inserting such APIs into the input sequence can be detected by analyzing the sample for irregular arguments or by searching for return values that are not utilized later during software execution. This weakness led us to **RQ1**, which explores alternative methods of creating adversarial examples without relying solely on feature insertion.

To answer **RQ1**, we explore the possibility of an API replacement attack that replaces API calls made by a malware with other functionally-equivalent API calls. This is a more restrictive attack as not all API calls can be replaced. Nonetheless, as all API calls in the sequences would have valid arguments and their return values utilized at some point, as they would have been before

replacement, the attack would be harder to detect. This chapter presents our attack by first describing our method for finding functionally equivalent API mappings and then discussing how our PSO-based attack utilizes those mapping.

## 5.1 Finding Functionally-Equivalent API Calls

To perform the AdversarialPSO attack on API-based malware detection models, we first must find functionally equivalent API calls to replace the API calls made by the malware. An obvious option for such functions is to use the low-level API calls the are eventually called by the high-levels APIs. For example, instead of using the WriteFile API call, we can use the NtWriteFile or ZwWriteFile. Although we do use low-level functions as potential replacements for their high-level counterparts, we conduct this study to find more equivalencies.

To find functionally-equivalent API calls, which we will refer to henceforth as *mappings*, we turn back to ML. In this dissertation, we focus on Windows malware as that is most common type of malware found in the wild. We therefore use the documentation provided by the Microsoft Developer Network (MSDN)[1] as a corpus for our ML techniques. MSDN documentation contains descriptions for most API calls supported by Windows 7. Using a web crawler, we were able to obtain the documentation for 867 API calls.

To facilitate code migration, researchers in the software engineering domain have explored methods to automate the process of finding function mappings between different platforms. For example. Gu et al. present DeepAM [20], a method that utilizes Seq2Seq, which is a ML technique that transforms one sequence to another, to find API mappings between Java and .Net. Another example is the work done by Bui and Jiang [10] that aims to learn cross-language representation to translate source code from one language to another. In their work, they translate code between Java and C# using Word2Vec, a neural network approach for creating word embeddings. This concept is further explored by Bui where the author aims to find mappings without depending on labeled datasets [9], unlike previous work which required some equivalent APIs between the platforms to be identified. The author uses unsupervised domain

[1] https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list

adaption via GANs to find the parallel APIs.

In finding equivalent mappings, our problem differs from that of related work. First, to the best of our knowledge, all related work that aims to find equivalent functions have access to both the functions' source code and documentation. In our problem however, we have partial access to documentation and we only have access to the functions' compiled code, but not their source code. The reason we have partial access to documentation is that many of the functions in our dataset consist of native low-level functions that are not made public by Windows. As for the functions' code, we access them through Dynamic Link Libraries (DLL)s. DLLs are compiled binaries and disassembling them provides assembly code, which is more complex and less informative than high-level source code. Another difference in our work is that while related work map functions across different platforms, we search for function mappings within the same platform.

We use unsupervised learning methods to find function mappings. Specifically, we use a combination of K-means, Latent Dirichlet Allocation (LDA), and Doc2Vec. Before we describe our approach for finding function mappings, we first provide some background on the unsupervised methods we use.

### 5.1.1 Background

In the following sections, we provide a brief overview on K-means, LDA, and Doc2Vec:

#### K-Means

K-means is an unsupervised learning approach that divides samples $X$ into $k$ disjoint clusters $C$ [44]. Each cluster is described by the mean $\mu_j$ of all the samples within the cluster, also known as the centroid. The clustering algorithm has three steps:

1. Randomly choose $k$ centroids, where $k$ is a hyperparameter that is specified beforehand.

2. Repeat

   (a) Assign samples to the nearest centroid.

(b) Update the centroids by calculating the mean of all samples they were assigned.

The process is repeated until the centroids no longer change or the changes are smaller than a certain threshold. The objective is to minimize the within-cluster sum of squares shown in Equation 5.1.

$$\sum_{i=0}^{n} \min_{\mu_j \in C} (\|x_i - \mu_j\|^2) \tag{5.1}$$

Another way to evaluate the quality of the clusters is using the Silhouette Score, which measures the cohesion and separation of the clusters [70]. Cohesion measures how similar samples are to samples of the same cluster and separation measures how similar they are to samples of other clusters. The range of the Silhouette Score is between -1 and 1, where higher numbers indicate better fits.

The inertia measure is used in the Elbow Method to determine the best $k$ for K-means. This is determined by plotting the inertia for each $k$ and observing the point where it starts to diminish. This will create an *elbow*-like shape indicating where the best $k$ would be. Both the Silhouette Score and Elbow Method are used in conjunction to select the best $k$ for K-means.

### LDA

LDA is a generative topic modelling algorithm that provides an explicit representation of a document by generating a finite mixture of topic probabilities [8]. Topic modelling is the task of finding the best topics that describe a set of documents. Essentially, the LDA algorithm considers each document as a collection of latent topics and individual words in the document as attributing to one of the topics. In other words, a document is assigned a set of topics based on the words in the document.

The LDA training process computes the latent variables $\theta$, $\zeta$ and $\varphi$, where $\theta$ is the topic-document distribution, $\zeta$ is the set of topics, and $\varphi$ is the word-topic distribution. It does so by using an alternating Expectation-Maximization (EM) procedure that empirically estimates the parameters. The Expectation step performs inference on a collection of documents (in this case, function documentation), and the maximization step accumulates the newly

acquired statistics from the E-step and updates the model. To infer topics from function documentation, we use the online variational Bayes (VB) method proposed by Hoffman et al. in [23] and implemented in the Gensim library[2].

#### Doc2Vec

To learn fixed-sized representations of variable-sized texts, Le and Mikolov [42] propose the unsupervised learning approach Paragraph Vector, also known as Doc2Vec. Doc2Vec extends Word2Vec to generate numerical representations for sentences or paragraph. Word2Vec, also known as word embedding, learns word representations by examining the context surrounding the word. The goal is to place words with similar context near each other in vector space. Both Word2Vec and Doc2Vec train neural networks to produce vector representations of the input.

### 5.1.2 Function Features

To find function mappings, we use 6 types of features extracted from both the documentation and code. We use a web scraper on `https://docs.microsoft.com` and `https://msdn.microsoft.com` to extract the following:

- Short description: One or two sentences that describe the API call's functionality. Often found at the top of the documentation page.
- Input parameters: Includes the type and name of each parameter used by the API call, whether the parameter is optional or required, and a short description for each parameter.
- Return type: The type of value returned by the function
- Required Libraries: The libraries required for the function to run
- Remarks: Additional information on the functionality and usage of the API call.

We also extract the functions' assembly code from a list of DLLs we compiled from the required libraries field. We disassemble the DLLs using IDA Pro and

[2] `https://radimrehurek.com/gensim/models/ldamodel.html`

extract the assembly code for each function in our API list. However, not all API calls had code as some functions were simply wrappers for others. The process of extracting function data is depicted in Figure 5.1.

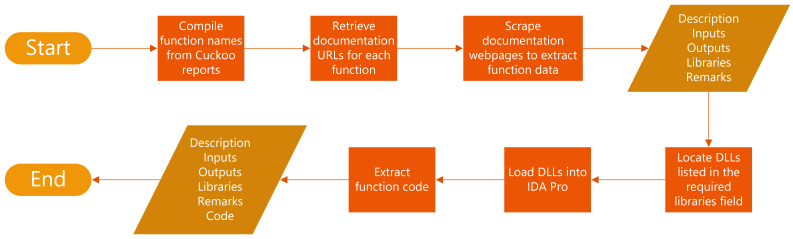


Figure 5.1: Overall Process for extracting API call data for finding function mappings

### 5.1.3 Unsupervised Learning for Finding Function Mappings

As discussed in the beginning of this chapter, generating adversarial examples from malware samples could be done by either inserting API calls into the malware's function call sequence, or by replacing the calls made by the malware with functionally-equivalent API calls. For the insertion method to retain the malwares' original functionality, the functions inserted must not alter the malwares' execution flow. Therefore, the API calls being inserted are limited to no-ops, which are calls that have no effect on the underlying system or the runtime software state. Such calls can be detected by scanning the malware samples for function calls with irregular arguments or unutilized return values. This weakness could be avoided by replacing the function calls with other functionally-equivalent calls, the method we propose in this dissertation.

To find function mappings, we use a hierarchical approach consisting of the three unsupervised learning techniques, K-means, LDA, and Doc2Vec. We apply these methods on the function documentation to find other functions with similar functionality. We also use the Doc2Vec approach on the functions' disassmebled code for further validation of equivalency. To verify our results, we manually replace a function calls from a subset of malware samples and execute those samples in a sandbox. We confirm that after replacing the calls, the malware samples still run as intended.

Essentially, our method for finding function mappings combines results

from the three aforementioned unsupervised learning techniques. If two functions reside within the same K-means cluster, share the same or several LDA topics, and have high cosine similarity between their Doc2Vec vector representations, we then consider them as candidates function mappings. As a final verification, we manually inspect those mappings to ensure that functionality is maintained.

All three methods are applied on the documentation corpus, but only Doc2Vec was used on the code. The reason is, extracting topics using LDA from assembly instructions could prove difficult due to the small vocabulary size of the code. Similarly, using K-means to divide samples into clusters based on disassmbled code would be ineffective for the same reason. However, we were able to quantify assembly code with Doc2Vec using long embedding vectors (100).

In the following section, we discuss our approach for finding function mappings:

**K-means**

To vectorize the terms in the corpus, we use Word2Vec on the 200 most salient terms in each document. We first preprocess the text by stemming all the terms and removing stop-words such as 'the', 'a', and 'in'. We then use Term Frequency-Inverse Document Frequency (TF-IDF) to select the 200 most prominent terms in each document. TF-IDF counts the number of times a term appears in a document, and then weights the importance of that term by counting its occurrence in other documents. A term that appears in many documents is less important than a term that appears in fewer documents. We consider uni-grams, bi-grams, and tri-grams when calculating the TF-IDF scores so that context is included in the calculations.

After selecting the top 200 grams, we train a Word2Vec model and use it to vectorize the corpus. The word embeddings generated by Word2Vec is used to train the K-means model. We empirically test multiple values of k (5-20) with different variations of documentation data. As shown in Figures 5.2 and 5.3, based on the Elbow Method and Silhouette Scores, 7 is the optimal number of clusters to use in K-means.

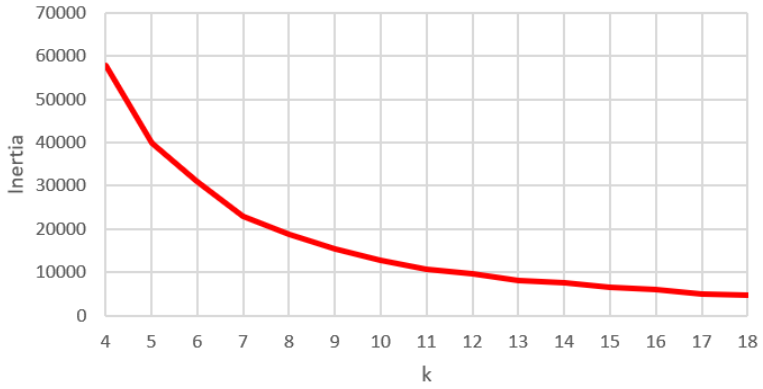


Figure 5.2: Elbow Method for determine best $k$ for K-means

### LDA

The second unsupervised method we use is LDA. First, we preprocess the data as we did in K-means by removing stop-words and by stemming all the terms in the corpus. We then tokenize the documents using Bag-of-Words (BOW) and train the LDA model. To evaluate the quality of topics, we use the Coherence Score, namely, the UMass measure. The UMass Coherence Score is measured as following:

$$SCORE_{UMass}(w_i, w_j) = log\frac{D(w_i, w_j) + 1}{D(w_i)} \tag{5.2}$$

Where $D(w_i, w_j)$ is the number of documents words $w_i$ and $w_j$ appear together and $w_i$ measure the number of documents $w_i$ appears alone. According to the UMass measure, if two words belong to the same topic, then it make sense that they appear frequently together and would thus achieve higher scores. The overall score is then calculated as:

$$Coherence = \sum_{i<j} score(w_i, w_j) \tag{5.3}$$

Using the Coherence Score, we selected the best number of topics to generate by re-training LDA to generate variable number of topics. As shown in Figure 5.4, 10 topics achieved the best Coherence Score (closer to 0 is better).

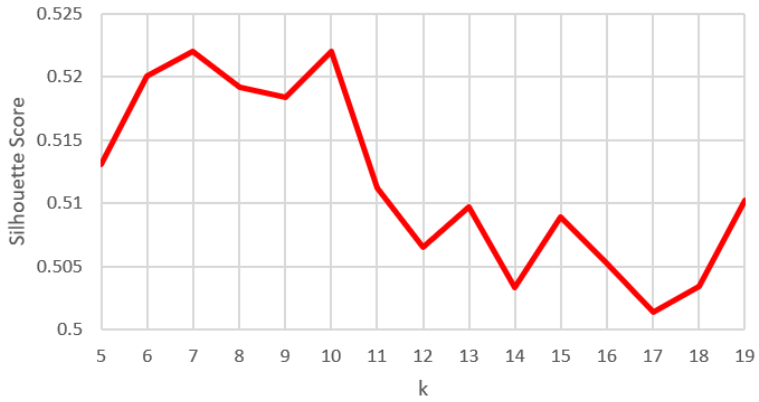


Figure 5.3: Silhouette Scores for different $k$'s for K-means

Once the model is trained with the optimal number of topics, we extract the topics that represent each function's documentation.

### Doc2Vec

The third and final unsupervised method we use is Doc2Vec. We use Doc2Vec to determine how similar two functions are according to both their descriptions and code. To do so, for the descriptions, we preprocess the corpus as we did with K-means and LDA. For the code however, no preprocessing was used as the data does not conform to natural language rules.

To extract the code for each function, we first retrieve the DLLs listed in the *Required Libraries* field. For most of the DLLs, a copy was found in the System folder of a fresh Windows 7 installation. We extract the code as is by including both operations and their operands. If jump instructions were present within the function scope and if the jumps lead to locations within the same DLLs, we would follow the jump instructions and extract the code at the jump destination.

We train two Doc2Vec models, one for function documentation and the other for function code. For both models, we use embedding vectors that are 100 dimensions long and we train them for 100 epochs each. We then use the models to generate embeddings for each function's documentation and code.

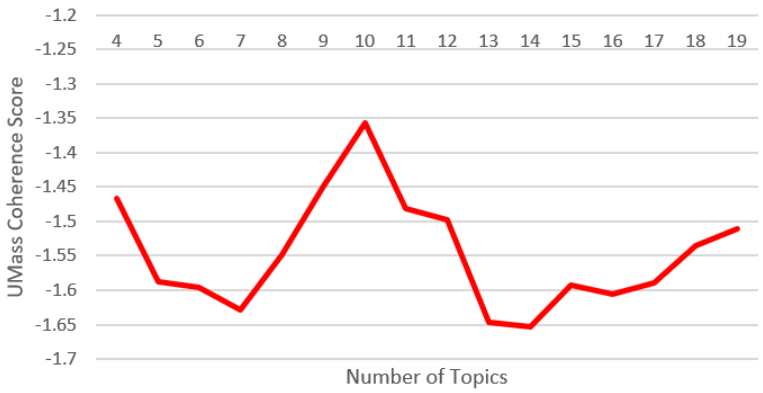


Figure 5.4: UMass Coherence Score for LDA

Finally, we use cosine similarity to measure the similarity between functions based on the embeddings generated by the Doc2Vec models.

### Combining Results

We use the results of all three methods to find function mappings. For each function, we iterate through all other functions within the same K-means cluster and compare the LDA topics for each two pairs. If the functions share 3 or more topics, we compare their Doc2Vec embeddings using cosine similarity. We use 0.6 as a threshold for both code and documentation to determine if two API calls are similar. Any two functions that meet the aforementioned criteria are evaluated further by manually inspecting their documentation and code to confirm their equivalency. Furthermore, for a subset of the mappings, we perform the replacements in a malware sample and execute it in a sandbox. We confirm that the malware still runs even after replacing the functions.

Of the 867 API calls considered in this study, we found mappings for 359 functions. For some APIs, multiple mappings were found. Table 5.1 shows a few examples of functionally equivalent API calls.

| API Call | Equivalent API |
|---|---|
| ZeroMemory | FillMemory |
| GetSecurityInfo | GetNamedSecurityInfo |
| EnterCriticalSection | TryEnterCriticalSection |
| GetParent | GetAncestor |
| SignalObjectAndWait | WaitForSingleObject |
| PeekNamedPipe | GetNamedPipeClientProcessID |
| CreateProcess | CreateProcessAsUser |
| DeleteMenu | DestroyMenu |
| GetWindowRect | GetClientRect |
| BringWindowToTop | SetWindowPos |

Table 5.1: Examples for functionally equivalent API calls

## 5.2 AdversarialPSO for Malware Detection Models

The mappings we generate in Section 5.1 create the search space for our AdversarialPSO attack. However, as discussed in Section 4.2, PSO cannot be directly applied on discrete data and must be adapted for particle movements. Although there are several approaches for using PSO on discrete data, we use an algorithm that works for the mappings we generated. In the next section, we discuss the implementation details for our AdversarialPSO attack that generates adversarial examples from malware.

### 5.2.1 Crafting Adversarial Malware using PSO

PSO variants that operate on discrete data assume either binary or fixed-sized features. In other words, particle positions are represented by an $dxm$ matrix where $d$ is the number of dimensions in the search space and $m$ is the number of possible values for each dimension. In the case of API replacement however, the number of available mappings are different from one API to another. Therefore, $m$ is different for each dimension, creating an irregular or jagged matrix. Furthermore, the shape of the matrix would be different from one malware sample to another as API calls would appear in different locations.

The approach we use is based on the binary PSO implementation proposed

by Kennedy et al. [34], which is extended by Veeramachaneni et al. to account for multi-valued discrete variables [82]. In this approach, the velocity vector calculated by Equation 4.2 is normalized to the [0,1] range using the sigmoid function $S_{id} = \frac{1}{1+e^{-V_{id}}}$. For each dimension, a random number is generated using a uniform distribution and is then compared against the normalized velocity to decide if the value in that position should be 0 or 1. Veermanachaneni et al. extend this approach by assuming discrete values between [0,m-1] and by normalizing the velocity using $S_{id} = \frac{m}{1+e^{-V_{id}}}$, which calculates a probability for each potential value in every element. Our approach combines both binary and multi-valued methods depending on the number of mappings that exist for an API.

With the above adjustments for discrete search spaces, the AdversarialPSO attack presented in Section 4.3.4 becomes as following:

### Initialization

As previously mentioned, the search process starts with initializing the particles by randomizing their positions in the search space. Particles are initialized by randomly perturbing an equal number of elements for each particle. For the malware attack, this is performed by first indexing all the API calls with available mappings and dividing those calls among the particles. Each particle would then probabilistically decide which elements to perturb. Essentially, for each particle, we create a $d$-length zero vector and assign a random number between [0,1] to each element in the particle's individual search space. This $d$-length vector is considered to be the particle's initial velocity. The elements are perturbed according to the initial velocity by randomly choosing one of the available mappings. In large swarms, each particle is assigned fewer elements, resulting in a more fine-grained search for adversarial examples. The malware-specific algorithm for the initialization stage can be seen in Algorithm 3.

### Optimization

The optimization step for the malware AdversarialPSO attack uses the same high-level operations shown in Section 4.3.4. However, as the attack operates in discrete space, there are extra steps for velocity calculations and particle movements. The velocity is calculated for each particle $i$ in every iteration $t$, and is computed based on particle's best position $p_i$ and the swarm's best

**Algorithm 3** Initializing the swarm for malware detection models

**Input:** input $x$, particle array $par$
$bestFitness \leftarrow 0$ # swarm-wide best
$bestPosition \leftarrow x$
$searchSpace \leftarrow$ indices of APIs with mappings
$n \leftarrow int(length(searchSpace)/P)$ # elements per particle
**for** $p$ *in* $par$ **do**
  $p.velocity \leftarrow [0] * length(x)$
  $elements \leftarrow$ select random $n$ elements from $searchSpace$
  remove $elements$ from $searchSpace$
  $p.position \leftarrow x$
  **for** $element$ *in* $elements$ **do**
    **if** $p.velocity[element] > random\ number$ **then**
      $perturb\ p.position[element]$ #select random mapping
    **end if**
  **end for**
  $fitness \leftarrow calculateFitness$ #includes update to $q$
  $p.bestFit \leftarrow fitness$
  $p.currentFit \leftarrow fitness$
  $push\ (p.bestfit, p.position)\ to\ p.pastPosition$
  $p.bestPos \leftarrow p.position$
  **if** $p.bestFit > bestFitness$ **then**
    $bestFitness \leftarrow p.bestFit$
    $bestPosition \leftarrow p.bestPos$
  **end if**
**end for**
**return** $par$, $bestPosition$, $bestFitness$

position $p_g$. For each particle, we compare the particle's current position $x_i$ against $p_i$ and $p_g$ to determine how to move the particle. For continuous data, we calculate the velocity using Equation 4.2, which directly subtracts $x_i$ from $p_i$ and $p_g$. For discrete data however, as subtracting is not possible, we rely on logical operators to compare the positions.

When calculating the velocity for a particle, we use an indicator vector to represent each element in $x_i$, $p_i$, and $p_g$. This length of this vector is equal to the number of potential API calls for that element. For example, if the $n$th element in $x_i$ is a WriteFile call, and WriteFile has two potential mappings, then the $n$th element in $x_i$ would be represented as [1,0,0], where 1 indicates which of the three calls is currently in the sequence. To compare $x_i$ against $p_i$ and $p_g$, for each element $n$ in $x_i$. we perform the following:

$$v_i^n(t+1) = wv_i^n(t) + c_1R_1((\neg x_i^n) \wedge p_g^n) + c_2R_2((\neg x_i^n) \wedge p_i^n) \tag{5.4}$$

Where $\neg$ is the logical *not* operator and $\wedge$ is the logical *AND* operator. Using the above Equation, multiple mappings could have a chance to be picked. If the same weights were used in $C1$ and $C2$, the two random numbers $r1$ and $r2$ would in this case control the odds for which mapping is chosen (the one used by $p_g$ or the one used by $p_i$). However, $C1$ or $C2$ could be used to favor swarm best over particle best, or vice versa. Finally, the mapping with the highest probability is compared against a random number and if it was found to be larger, the particle replaces its current API with that mapping.

As explained in Chapter 4, we implement a random mutation operation to help avoid getting stuck in local minimas. This is implemented by specifying a small probability (0.1) that a particle randomly changes some of the mappings.

### 5.2.2 Evaluation

To evaluate the attack, we use the model we describe in Chapter 3. As the field of dynamic-based malware detection suffers from a lack of benchmarks and publicly available work, we resort to creating our own benchmark for evaluation. Also, at the time of writing, related work was limited to approaches that only add to the malware call sequences. These approaches are vulnerable to detection using non-ML techniques. Our work however, replaces API calls with other functionally-equivalent APIs. Due to the limited number of available mappings, this attack is launched under more restrictive constraints

**Algorithm 4** Optimization step for the malware AdversarialPSO attack

**Input:** particle array *par*, swarm-wide best fitness $bestFitness$, mutation probability $m$ and swarm-wide best position $bestPosition$
**for** $p$ *in par* **do**
  **if** $m >$ *random number* **then**
    **for** *element in p.position* **do**
      *perturb p.position*[*element*] #select random mapping
    **end for**
  **end if**
  **for** *element in p.position* **do**
    **if** *mapping for element exists* **then**
      calculate velocity for element # Eq. 5.4
      **if** $max(p.velocity[element]) >$ *random number* **then**
        $p.position[element] \leftarrow updatePosition$
      **end if**
    **end if**
  **end for**
  $fitness \leftarrow calculateFitness$ #includes update to $q$
  $p.currentFit \leftarrow fitness$
  **if** $fitness > p.bestFitness$ **then**
    $p.bestFit \leftarrow fitness$
    $p.bestPos \leftarrow p.position$
  **end if**
  **if** $p.bestFit > bestFitness$ **then**
    $bestFitness \leftarrow p.bestFit$
    $bestPosition \leftarrow p.bestPos$
  **end if**
**end for**
**return** $bestPosition$

and is bound by the mappings available to the attack. Therefore, comparing the two types of attacks would not be indicative of how the replacement-based attack performs in comparison to the other attacks.

To evaluate the black-box attack, we use the same mappings to launch a white-box gradient-based attack. The goal of using the white-box attack as a baseline is twofold: first, it provides means to evaluate the black-box aspect of the attack by comparing it to a white-box attack that shares the same constraints, and second, it provides insight on which future direction to pursue to enhance the attack. Before reporting the results of AdversarialPSO, we will first describe the white-box gradient-based attack we use as a baseline. We will then discuss our evaluation of the AdversarialPSO attack and provide some possible avenues for future work.

**Gradient-based White-box Attack**

The gradient-based white-box attack we launch is an extension to the work done by Grosse et al. in [18], which is based on the JSMA attack proposed by Papernot et al. [62]. Essentially, the attack computes the gradient of the loss function with respect to the input and makes changes that maximizes the loss according to the gradient. Formally, the jacobian matrix of the loss is computed as:

$$J_F = \frac{\partial F(X)}{\partial X} = \left[\frac{\partial F_i(X_j)}{\partial X_j}\right]_{i\in[0,1],j\in[1,n]} \tag{5.5}$$

Where $F$ is the neural network, $X$ is the input, $i$ is the class label (0 for benign and 1 for malicious), and $j$ is the feature in input $X$.

In their attack, Grosse et al. search for perturbations in $X$ that maximizes the classification of the input as benign. However, they only search for positive changes (flipping 0 to 1) to maintain malware functionality. In our attack however, we search for changes that maximize the classification to benign through available API mappings. To do so, as some API calls have multiple mappings, we first compute the gradient for all available alternative APIs and then replace features in the input with the mappings that maximize the loss with respect to the input.

**Results**

We perform the evaluation using 183 correctly classified malware samples from the test set. The model we attack achieved an accuracy of 88.20% on the test set with 91.45% recall and 86.25% precision. On the 183 malware samples, the model is 99.36% confident in its predictions on average. The model was trained on the first 2500 API calls made a sample while retaining at most 3 consecutive calls of the same API in case of duplication. In the malware test set, of the 2500 API calls used by the model, we had on average 553.69 APIs with available mappings with an average of 1.46 mappings per API. Essentially, the majority of the API calls had only 1 mapping, however, some API calls had more (e.g. MapViewOfFile could be replaced with MapViewOfFileEx or MapViewOfFileExNuma).

Interesting, as shown in Table 5.2, the AdversarialPSO black-box attack outperformed the white-box attack. This could be attributed to the susceptibility of gradient-based attacks to getting stuck in local minimas. PSO however, does not suffer from this due to the mutation operation which randomizes particle positions. The AdversarialPSO attack created 51 adversarial examples from the test set of 183 malware, achieving a 27.86% successful rate.. The gradient-based attack on the other hand, created adversarial examples for 13.66% of the test set (25 samples). However, when it did succeed, the gradient-based attack would find the adversarial example much faster than the AdversarialPSO attack. That is because is it guided by the gradient and is able to find the solution after only a few steps. The average iterations for the black-box attack was 358, which amounts to an average of 2688 queries made to the model. The white-box attack on the other hand, finds the adversarial examples within the first 5 iterations (when the gradient for all the mappings are computed and compared), where on average, a solution is found in 1.36 iterations. Although number of queries are not significant in white-box attacks (because attacks are assumed to have access to the model), on average, the white-box attack queried the model 5.08 times. Each time the attack computes the model gradient as shown in Equation 5.5, it counts as a single query. Also, when the attack classifies the sample to check if the label changed after perturbing the inputs, that also counts as a single query.

As both attacks were only able to generate adversarial examples for a small subset of the malware, this is a strong indication that the mappings set an upper-bound on how well the attacks can perform. Nonetheless, the ability

| Attack | Success Rate | Average Queries | Average Iterations |
|---|---|---|---|
| AdversarialPSO | 27.86% | 2688 | 358 |
| Gradient-based attack | 13.66% | 5.08 | 1.36 |

Table 5.2: Comparing black-box PSO-based attack against white-box gradient-based attacks

of AdversarialPSO to create more adversarial examples as the gradient-based attack while using the same constraints, demonstrates its ability to succeed in more restrictive setting where particles are limited in movements. To further evaluate its abilities, we test the attack on the image classification domain where such limitations do not exist. As we show in Chapter 6, AdversarialPSO performs comparably to the state-of-the-art in adversarial ML for image classification models.

### 5.2.3 Discussion

#### Mappings

As shown by the results in Section 5.2.2 and as we will shown in the next chapter, PSO is able to generate adversarial examples in a black-box setting. The attack on the malware detection model using replacement does generate fewer adversarial examples than attacks that insert features, however, replacing API calls is more robust against non-ML detection techniques than inserting null APIs. Also, the approach we use in this dissertation focused on finding 1-to-1 mappings, which produced a rather limited set of possible API substitutions. This could be extended by searching for 1-to-n, n-to-1, and n-to-n mappings. Essentially, any augmentation to the set of all possible changes would allow the particles to move more freely in the search space, which would in turn increase the chances of finding adversarial inputs. Finding more API mappings is possible direction for future work to enhance the API-replacement attack.

#### Attacking Static-based Malware Detection Models

The attack we present in this dissertation targets ML models that classify software as malicious or benign based on their behavior. Specifically, it targets models that use API call sequences that are recorded during software

execution. Another approach for malware detection is to train ML models on static-based features such as the opcode sequences found in the binaries [72] or their byte representations [66]. Similar to API calls, opcode instructions could be mapped to functionally-equivalent sequences that perform the same operations. In fact, this concept has been exploited by metamorphic engines for quite some time to evade signature-based detection. The perform these mutations by reordering instructions, replacing instructions, and renaming variables. As the goal of these metamorphic engines is to evade signature-based detection, they are largely ineffective against ML approaches that learn patterns for classification. However, the attack we propose in this dissertation could be adapted to operate on static-based features, the same ones used by metamorphic engines, to create adversarial examples that evades the detection of ML models. This would essentially be creating the next generation of metamorphic engines that use intelligence to fight intelligence. This is an avenue that we certainly plan to explore in the future.

# Chapter 6

# Attacking Image Classification Models using PSO

In this chapter, we describe how we use PSO to generate adversarial examples for image classification models. The work we describe in this chapter aims to answer our second research question **RQ2**, which explores the possibility of re-adapting the AdversarialPSO attack to work in the image classification domain. Evaluating the attack in a different domain with different constraints would provide insight on the limits of the attack. If AdversarialPSO had low success rates against images while sharing the same constraints as other attacks in this area, this would show that the attack itself is weak, on top of the possible limitation introduced by the limited API mappings. However, if the attack performed well against image classification models, that would prove that AdversarialPSO is able to effectively generate adversarial examples but was limited by the amount of available mappings in the replacement-based malware attack.

To compare our work against baselines in the image classification domain, we adhere to the constraints of the field, which differ from the constraints used when attacking malware detection models. Specifically, when attacking images, the amount of perturbations added to the inputs are bound by a pre-defined $L_\infty$ limit that must be maintained for a fair comparison with the related work. $L_\infty$ measures the maximum change to any of the coordinates,

where $L_\infty = max(|x_1 - x'_1|, |x_2 - x'_2|, \ldots, |x_d - x'_d|)$. This ensures that perturbations are not too large that the semantic properties of the images are lost. For each of the benchmark datasets we consider in this work, we use the same $L_\infty$ used in the baseline. To control the perturbations added to the input image, we define an upper bound value $B$ of maximum change to limit the $L_\infty$ distance between the adversarial image and the original image. Essentially, we use the clip operator such that $x^{'} = clip(x_i + v_i, x_i - B, x_i + B)$. Additionally, we also apply box constraints to maintain valid image values when adding perturbations. These constraints are applied to Equation 4.1 to yield:

$$x_i(t+1) = clip(clip(x_i(t) + v_i(t+1), x_i - B, x_i + B), 0, 1) \tag{6.1}$$

## 6.1 Operations Specific to Image Classification

Although the overall process for generating adversarial examples is similar for both malware and images, we implement additional operations specific to images to enhance the efficacy of the attack.

### 6.1.1 Block-Based Perturbation

Similar to related work [52] [29] [6], we exploit the spatial regularity of adjacent pixels by splitting the input into blocks and perturbing all the pixels in each block en masse.

### 6.1.2 Following the Edge of the $L_\infty$ Ball

As observed by Moon et al. in [52], the optimal solution when crafting adversarial examples often reside at the edges of the $L_\infty$ ball. Based on this observation, when initializing and randomizing particles, we set their positions at the edge of the $L_\infty$ ball to observe the highest (or lowest) fitness for each dimension. Particles are then moved inwards using Equations 4.2 and 6.1. Moving inwards from the edge ensures that particles get enough velocity to reach the other end quickly if the opposite position was found to have better fitness. Otherwise, particles would waste queries moving around the center of the ball until they eventually build enough velocity towards the position with the highest fitness.

### 6.1.3 Redundancy Minimization

Since we seek to minimize the number of queries, it is helpful to avoid redundantly attempting the same modifications to the image across multiple particles. Instead, we can leverage exploration in the PSO algorithm to benefit from the best position in the swarm when such a modification is found to be useful. We thus propose a novel method to minimize unnecessary queries from redundant checks on already perturbed blocks. Essentially, if one of the particles has modified one of the blocks in a given way, e.g. it increased the red channel on all pixels in that block, then we prevent other particles from making the same modification. To do this, we first define a set $\beta$ with all *available* blocks (which are still eligible to be modified), $\beta = (b_1, b_2, b_3, \ldots, b_n)$.

Then, for each block in the set, we create a list of all possible directions containing the positive and negative directions for each channel in the block. For grayscale images, which contain only a single channel, the list of possible channel directions $cd$ is given by $cd = \{(1), (-1)\}$. For RGB images, it is

$$cd = \{(1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), (0,0,-1)\}.$$

In other words, any single channel could be increased or decreased. When a direction in a block is assigned to a particle, that direction is then removed from the list to avoid multiple particles perturbing the same block in the same direction. When all the directions in a block are assigned to particles, we remove that block from the set $\beta$. When there are no more blocks in the set, we increase the granularity of the blocks by dividing the block-size by half and recreate the block set to contain the smaller blocks.

Each particle maintains a list of all the blocks and directions assigned to it. This list is used to avoid assigning an opposite direction to the particle which would cancel out a direction that it was previously assigned.

With these modifications in place, the AdversarialPSO attack on image classification model becomes as following:

### 6.1.4 Initialization

Particles are initialized by randomly assigning an equal number of blocks to each particle. Two hyperparameters control how the swarm is initialized: the number of particles in the swarm $P$ and the initial block-size $b$, which determines the number of initial blocks created and the number of blocks

assigned to each particle. Each particle begins with the input image $x$ and the set of blocks $\beta$ with a single direction for each block. Particles are then dispersed in the search space by perturbing all the blocks assigned to them to the edge of the $L_\infty$ ball according to the directions they were given. Once the particles are created and dispersed, their fitness is calculated and subsequently used in the optimization step.

Changes to the initialization process can be seen in Algorithm 5.

### 6.1.5 Randomization

In every iteration, in addition to particle movements, each particle is assigned the next set of blocks and directions as was done in the initialization stage. This randomization is performed after the particles are moved according to their calculated velocity vectors to allow the exploration of additional regions of the search space. This randomization process is performed until all the directions in all the blocks are assigned to a particle, at which point, the granularity of the blocks are increased, and the particles are re-initialized with the swarm best position as a starting point. Re-initializing the particles resets their best positions, which would otherwise cause them to retract to the previous granularity in the next iteration. The algorithm for the randomization process can be seen in Algorithm 6.

### 6.1.6 Reversal

After all the blocks are assigned to particles, a reversal of all the movements that have caused a negative impact on the fitness is performed. This is done before increasing the granularity of the blocks. The reversal is performed on the swarm best position by iterating through the past positions of each particle and applying an opposite step for any movement that caused a negative fitness for the particle. Moon et al. perform a similar operation by alternating between adding perturbations and removing perturbations [52]. Their reasoning however is due to submodularity, which refers to the diminishing return affect that occurs when the set of all perturbations increase in size. In our implementation however, instead of removing the perturbation from the swarm best position, we perturb in the opposite direction with the idea that if a direction decreased fitness, the opposite direction would increase it. We find

**Algorithm 5** Initializing the swarm

**Input:** input image $x$, particle array $par$, block-set $B$, and maximum change $m$.
$bestFitness \leftarrow 0$ # swarm-wide best
$bestPosition \leftarrow x$
$n \leftarrow int(length(B)/P)$ # blocks per particle
**for** $p$ *in* $par$ **do**
  $blocks \leftarrow$ select random $n$ elements from $B$
  $p.position \leftarrow x$
  **for** $block$ *in* $blocks$ **do**
    *push* $block$ *to* $p.blockList$
    $direction \leftarrow$ select random direction from $B[block]$
    *pop* $direction$ $from$ $B[block]$
    *push* $direction$ *to* $p.blockList[block]$
    **for** $i$ *in* $block$ **do**
      $p.position_i \leftarrow p.position_i + m * direction$
    **end for**
  **end for**
  $fitness \leftarrow calculateFitness$ #includes update to $q$
  $p.bestFit \leftarrow fitness$
  $p.currentFit \leftarrow fitness$
  *push* $(p.bestfit, p.position)$ *to* $p.pastPosition$
  $p.bestPos \leftarrow p.position$
  **if** $p.bestFit > bestFitness$ **then**
    $bestFitness \leftarrow p.bestFit$
    $bestPosition \leftarrow p.bestPos$
  **end if**
**end for**
**return** $par$, $bestPosition$, $bestFitness$

**Algorithm 6** Randomize Particles

**Input:** particle list $par$, block-set $B$, change rate $cr$, and maximum change $m$.
**for** $p$ *in* $par$ **do**
  **if** $B$ is not empty **then**
    $blocks \leftarrow$ select random $cr$ elements from $B$
    **for** $block$ *in* $blocks$ **do**
      **if** $block$ *in* $p.blockList$ **then**
        $d \leftarrow p.blockList[block]$
        $direction \leftarrow$select random direction from $B[block] - \{d, -d\}$
        *pop* $direction$ *from* $B[block]$
        *push* $direction$ *to* $p.blockList[block]$
        **for** $i$ *in* $block$ **do**
          $p.position_i \leftarrow p.position_i + m * direction$
        **end for**
      **else**
        *push* $block$ *to* $p.blockList$
        $direction \leftarrow$ select random direction from $B[block]$
        *pop* $direction$ *from* $B[block]$
        *push* $direction$ *to* $p.blockList[block]$
        **for** $i$ *in* $block$ **do**
          $p.position_i \leftarrow p.position_i + m * direction$
        **end for**
      **end if**
    **end for**
    Compare new fitness against particle best and swarm best
  **end if**
**end for**

that in many instances that is, in fact, the case and a better position is often found. The algorithm for the reversal operation can be seen in Algorithm 7

**Algorithm 7** Reverse movements with negative fitness

```
1: Input: best position bestPosition and Particle list par
2: for p in par do
3:   for pastPosition in p.pastPosition do
4:     if pastPosition.fitness < 0 then
5:       bestPosition ← bestPosition − pastPosition.position
6:       if Fitness did not improve then
7:         Undo last changes
8:       end if
9:       pop pastPosition from p.pastPosition
10:    end if
11:  end for
12: end for
13: return bestPosition
```

The overall algorithm for generating adversarial examples from images can be seen in Algorithm 8.

## 6.2 Evaluation

### 6.2.1 Setup

To evaluate AdversarialPSO, we consider the the success rate (i.e., the ratio of successfully generated adversarial examples over the total number of samples) and the average number of queries needed to generate adversarial examples. We compare our results against the Parsimonious Black-Box Adversarial Attack [52], NES [29] and Bandits [30] using the benchmark dataset Imagenet. Similar to the related work, we evaluate the attack using InceptionV3. The Imagenet results for both untargeted and targeted attacks are obtained from running AdversarialPSO on 1,000 correctly classified samples from the indices list provided in the Parsimonious attack. The same target labels used in [52] and [30] are used for the targeted experiment. The results for the untargeted and targeted Imagenet attacks are reported in Sections 6.2.3 and 6.2.4,

**Algorithm 8** Optimization

**Input:** maximum queries $q_{max}$,block-set $B$
**while** $q < q_{max}$ **do**
  **if** *Success* **then**
    **return** *bestPosition*
  **end if**
  *Move Particles*
  **if** $B$ *is empty* **then**
    $performReversal$
    $increaseGranularity$
    $initializeParticles(bestPosition)$
  **else**
    $randomizeParticles$
  **end if**
**end while**
**return** *bestPosition*

respectively.

We compare the AdversatialPSO attack on MNIST and CIFAR-10 against the approach used by Bhagoji et al. [6] to show the improvements attained from our modifications to the PSO algorithm. Similar to the models used in [6], we use ResNet-32 and a 2 layer Convolutional Neural Network for CIFAR-10 and MNIST respectively. Furthermore, we use the same $L_\infty$ limits, where the MNIST attack was set to 0.3 and CIFAR-10 was set to 0.03137255 (equivalent to 8 out of 255 in the per-pixel range). In contrast however, we only use 5 particles in our implementation whereas Bhagoji et al. used 100. As we show in Section 6.2.2, we achieve higher success rates with much smaller swarms. For all MNIST evaluations, we use an initial block-size of 2 without requiring to increase the granularity. For CIFAR-10, we use an initial block-size of 8.

To test the wide-applicability of the attack, we evaluate its effectiveness on two additional models for CIFAR-10 and MNIST. We first evaluate the attacks on the same models used in [2, 11, 12]; we refer the readers to Carlini and Wagner's paper [11] for more details. For CIFAR-10, we also test the attack on a CNN-Capsule model trained with data augmentation as described by Sabour et al. [71], which achieves a test set accuracy of 82.43%. Further-

more, we evaluate AdversarialPSO on MNIST using a Hierarchical Recurrent Neural Network [43] with an accuracy of 98.64%. We also test the attack on an adversarially trained CIFAR-10 ResNet classier as was done in [52], by using the same pretrained network provided by MadryLab[1]. The results for the wide-applicability test, including the attack on the adversarially trained CIFAR-10 ResNet model, are reported in Section 6.2.5

To demonstrate the effect of using larger swarms on the generated adversarial examples, we re-run the untargeted Imagenet experiment with differently-sized swarms. We report the average per-pixel $L_2$ distance between input images and their adversarial counterparts. With the increase in granularity when using more particles, we show how larger swarms produce adversarial examples with a lower $L_2$ average. We show the results of this analysis in Section 6.2.6.

### 6.2.2 Untargeted MNIST and CIFAR-10

To demonstrate the effectiveness of AdversarialPSO, we compare our attack against the approach used by Bhagoji et al. [6]. As shown in Tables 6.1 and 6.2, AdversarialPSO not only outperforms the standard PSO used by Bhagoji et al., it also outperforms the GE approach used by the authors. For MNIST, the only approach to have a higher success-rate is the Iterative Finite Difference attack at 100%, however the average number of queries was above 60K. In our implementation, we set a maximum budget of 10K queries, therefore for many of the samples that was deemed successful in the Iterative Finite Difference attack, we would have considered as failed in our implementation for passing the query budget.

Regarding the average $L_2$, using a swarm with 5 particles produces adversarial examples with comparable distances. However, by increasing the number of particles in the swarm, better quality adversarial examples could be generated at the expense of more queries. Repeating the same experiment but with 10 particles produces an average $L_2$ of 4.9, but with an average of 296 queries.

Similarly for CIFAR-10, the only two approaches to have higher success rates are the iterative GE and iterative Finite Difference, both of which however, have much higher query averages. As previously mentioned, we use a

[1] `https://github.com/MadryLab/cifar10_challenge`

query budget of 10,000 queries and accordingly, for many of the samples that have been considered successfully attacked in the iterative GE and iterative Finite Difference attacks, we would have considered to have failed.

In examining the failed instances of the CIFAR-10 ResNet-32 model, we find that samples that failed were resistant to small perturbations. Particle movements had a low impact on the model's confidence scores and as such, executed for a large number of iterations until the the query budget was exhausted. Interestingly, for a majority of the samples however, the adversarial examples were crafted rather quickly without using much queries. This makes us wonder about the position of the failed instances in the search space with respect to the decision boundary. We believe those instances were located so far away from the decision boundary such that large changes were required for them to be misclassified. Figure 6.1 shows randomly chosen examples of attacks on both CIFAR-10 and MNIST.

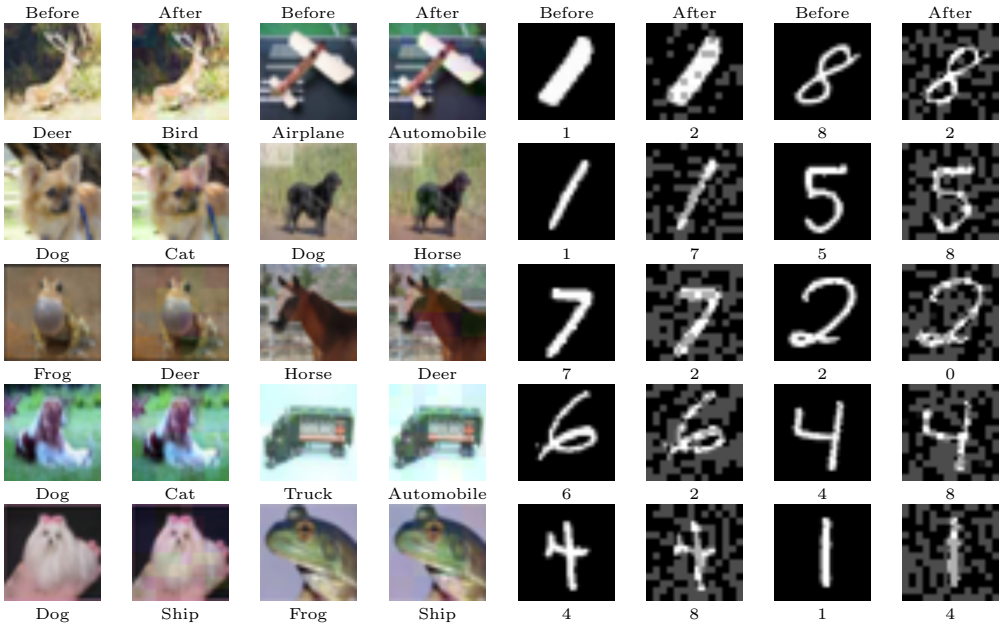


Figure 6.1: Untargeted attack using AdversarialPSO on MNIST and CIFAR-10

| | Success Rate | Avg. L2 | Avg. Queries |
|---|---|---|---|
| Finite Diff | 86% | 410.3 | 6144 |
| GE | 66.8% | 402.7 | 768 |
| Iterative Finite Diff | **100%** | 65.7 | 61440 |
| Iterative GE | 99.0% | 80.5 | 7680 |
| Their PSO | 89.2% | 262.3 | 7700 |
| SPSA | 88.0% | **44.4** | 7680 |
| AdversarialPSO | 94.9% | 336 | **129** |

Table 6.1: Results comparison: Untargeted attack on CIFAR-10 against the PSO and GE attacks of Bhagoji et al. [6]. The results we list for the Bhagoji attacks are obtained from their paper

| | Success Rate | Avg. L2 | Avg. Queries |
|---|---|---|---|
| Finite Diff | 92.9% | 6.1 | 1568 |
| GE | 61.5% | 6.0 | 196 |
| Iterative Finite Diff | **100%** | 2.1 | 62720 |
| Iterative GE | 98.4% | **1.9** | 8000 |
| Their PSO | 84.1% | 5.3 | 10000 |
| SPSA | 96.7% | 3.9 | 8000 |
| AdversarialPSO | 98.5% | 5.3 | **183** |

Table 6.2: Results comparison: Untargeted attack MNIST against the PSO and GE attacks of Bhagoji et al. [6]. The results we list for the Bhagoji attacks are obtained from their paper

### 6.2.3 Untargeted Imagenet

To evaluate the attack on the Imagenet dataset, we use the InceptionV3 model provided by Keras[2]. As per the Keras implementation, inputs are scaled to [-1,1], we therefore set the $L_\infty$ bound to 0.1 (equivalent to the 0.05 used by related work). We choose the first 1000 samples from the indices list found in the Parsimonious Black-Box Attack GitHub page[3] and attack each sample with a query budget of 10,000 queries. We also use 32 for an initial block-size,

[2] `https://keras.io/applications/\#inceptionv3`

[3] `https://github.com/snu-mllab/parsimonious-blackbox-attack`

similar to [52] and 10 particles in the swarm. The results for the untargeted attack on Imagenet are shown in Table 6.3 and Figure 6.2 shows randomly chosen examples of the images generated from the attack. As shown in the table, our attack achieves comparable success rates and number of queries as the related work, but with the advantage of providing controllable trade-offs between the number of queries and the quality of the adversarial examples.

| Attack | Success Rate | Avg. Queries |
|---|---|---|
| NES | 80.3% | 1660 |
| Bandits | 94.9% | 1030 |
| Parsimonious attack | **98.5%** | **722** |
| AdversarialPSO | 96.9% | 837 |

Table 6.3: Untargeted attack on Imagenet

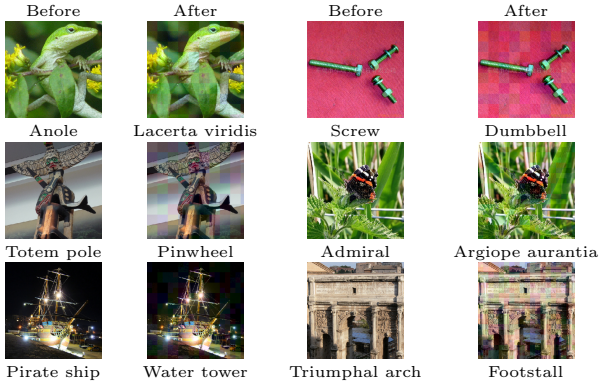


Figure 6.2: Untargeted attacks using AdversarialPSO on InceptionV3

### 6.2.4 Targeted Imagenet

To evaluate AdversarialPSO in a targeted attack, we use samples from the Parsimonious Black-box Attack's list of sample indices and we use the same labels as in [52]. Furthermore, similar to [52], we use an initial block-size of 32 and a query budget of 100,000 queries. We also use 10 particles as was done in the untargeted attack on Imagenet. Table 6.4 summarizes our results and

Figure 6.4 shows randomly chosen examples of the attack.

| Attack | Success Rate | Avg. Queries |
|---|---|---|
| NES | 99.7% | 16284 |
| Bandits | 92.3% | 26421 |
| Parsimonious attack | **99.9%** | **7485** |
| AdversarialPSO | 98.6% | 14959 |

Table 6.4: Targeted attack on Imagenet

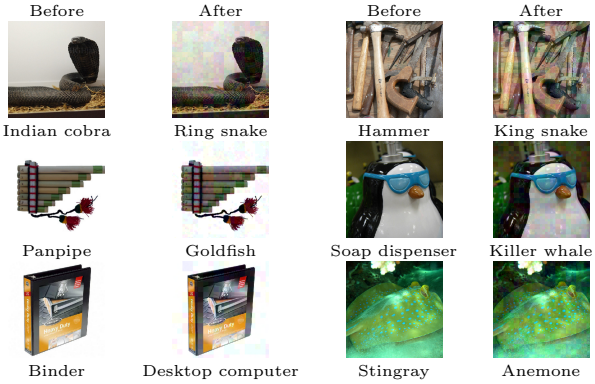


Figure 6.3: Targeted attacks using AdversarialPSO on InceptionV3

### 6.2.5 Wide Applicability of AdversarialPSO

As shown in Table 6.5, AdversarialPSO is effective on different architectures as the attack was able to generate adversarial examples with high success rates and low queries. The lowest success rate in this evaluation occurred when attacking the CNN-Capsule, or CapsNet, which incorporates random data augmentation in the training process. During the attack, we noticed that for this model, perturbations had less of an affect on the model's output, which caused the attack to exhaust the query budget for 10% of the samples. Testing the attack on CNN-Capsule without data augmentation, the success rate went up to 94.1% with the average queries being 199, which shows that training with augmented samples does help in making the model more robust

against perturbations.

| | MNIST | | |
|---|---|---|---|
| Model | Success Rate | Avg. L2 | Avg. Queries |
| C&W | 99.0% | 5.3 | 172 |
| HRNN | 98.46% | 3.9 | 30 |
| | CIFAR-10 | | |
| Model | Success Rate | Avg. L2 | Avg. Queries |
| C&W | 94.0% | 1.4 | 332 |
| CNN-Capsule | 90.0% | 1.3 | 214 |

Table 6.5: AdversarialPSO Wide-Applicability Evaluation Results

To test the attack against defended models, we evaluate AdversarialPSO against the adversarially trained CIFAR-10 model provided by MadryLabs. We use the same samples, $L_\infty$ bound, and query budgets as used in [52]. The results are shown in Table 6.6.

| Attack | Success Rate | Avg. Queries |
|---|---|---|
| NES | 29.5% | 2872 |
| Bandits | 38.6% | 1877 |
| Parsimonious attack | **48%** | **1261** |
| AdversarialPSO | 45.4% | 2341 |

Table 6.6: Untargeted attack on adversarially trained CIFAR-10 ResNet classier

As seen in the table, AdversarialPSO outperforms both Bandits and NES in terms of success rate and average number of queries. Although the Parsimonious Black-box attack remains the highest in success rate, AdversarialPSO performs comparably with the added advantage of providing a trade-off between queries and $L_2$.

### 6.2.6 Swarm-size Analysis

By re-running the untargeted Imagenet attack using swarms with different sizes, we show that increasing the number of particles lowers the average $L_2$

at the expense of more queries. The results are based on samples that were successfully attacked by all swarm sizes. As shown in Figure 6.4, there is a 15% improvement in adversarial example quality when increasing the number of particles from 5 to 20, at the cost of using 33% more queries. With this trade-off, an attacker that favors adversarial example quality over number of queries can use larger swarms. On the other hand, if fewer queries is more important to the attacker, then smaller swarms would be more beneficial.

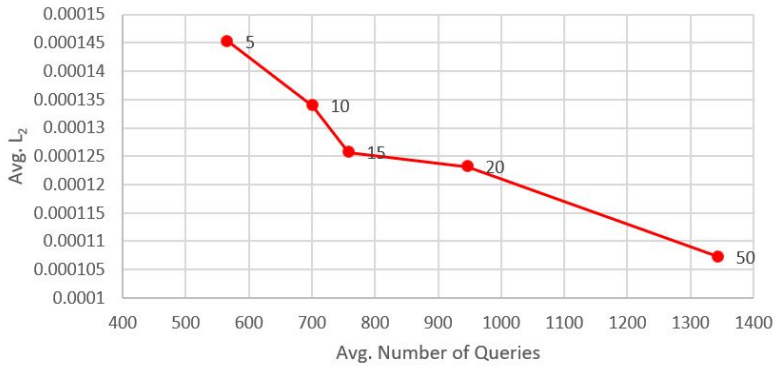


Figure 6.4: The effect of swarm size on the average number of queries and per-pixel $L_2$ distance. In the figure, the x-axis represents the number of queries, the y-axis represents the per-pixel $L_2$, and the number of particles are shown by the markers

## 6.3 Discussion

### 6.3.1 Potential Improvements

Our current implementation of the AdversarialPSO attack executes sequentially, where particles are looped through and moved one by one. This design choice was made to evaluate the speed of the attacks with the least amount of resources. A natural next step is to parallelize the process by dividing the particles among multiple CPU or GPU cores, allowing for faster execution of bigger swarms. It would not, however, affect either the $L_2$ difference between the adversarial examples and their inputs nor the number of queries submitted

to the target if the swarm size remains unchanged.

An extension to PSO is Multi-Swarm Optimization (MSO), which uses multiple sub-swarms instead of a single swarm. Using MSO with different starting points could help cover more areas of the search space and eventually find better adversarial examples; it, however, would add to the total number of queries if each MSO swarm is the same size as the PSO swarm. Alternatively, smaller sub-swarms can be used in MSO so that the total number of particles remains unchanged.

Having multiple sub-swarms also allows the use of different fitness functions or different swarm configurations. The sub-swarms could apply different distance penalties, mutate particles more or mutate them less, have different step sizes or have different exploitation/exploration/inertia weights. Sharing the same starting point but with different configurations could allow better exploration of different regions of the search space.

### 6.3.2 Limitations of Gradient-Based Attacks

Launching gradient-based black-box attacks could be made with or without using a local surrogate; both approaches have their limitations. First and foremost, when using a local surrogate that approximates the target, the success of the attack depends on the precision and efficacy of several intermediary steps that could affect the overall attack. The attacker must have a tight approximation of the target trained locally, and the adversarial examples must successfully transfer from the surrogate to the remote target. As demonstrated in [12], transfer-based attacks achieve low success rates when training on a surrogate in a black-box setting. This could be attributed to a poorly trained local surrogate that produces adversarial examples incapable of transferring to the remote target.

Alternatively, a gradient-based black-box attack could be launched directly on the target by estimating the gradients as was done by ZOO, NES, and Bandits. Although the attacks do generate high-quality adversarial examples, they would either require a large number of queries to do so or would be mostly dependable on the hyperparameter values chosen for the attack. Submitting multitudes of queries to the target is impractical in a real-world attack, as that would easily be noticed by an operator that is monitoring the volume of incoming queries. Furthermore, depending on multiple hyperparameters could

require tuning, which might not be possible in a real-world attack. The PSO approach thus appears to offer a better trade-off in this setting, where it more quickly converges to high-quality but non-optimal examples. Considering that a black-box scenario is a more likely setting, particularly in security-sensitive settings, attacks must be able to generate adversarial examples with a realistic number of queries without requiring tuning the attack to the target.

### 6.3.3 Large-Scale Adversarial Attacks

The AdversarialPSO attack we present in this dissertation provides the opportunity for attackers to launch large-scale attacks on remote targets that host model for high-dimensional datasets. Due to the nature of the attack, it can be easily scaled not only on a single machine by increasing particles, but across multiple machines that share a common target. We anticipate future attacks on machine learning models not to be launched from a single source but from multiple sources working in conjunction. In a sense, it is similar to botnets, where a large group of zombie bots are utilized for a single purpose. The AdversarialPSO attack provides the foundation for such attacks, where tens of thousands of particles can be launched by multiple attackers. Note that PSO does not require a GPU for efficient computation, making it more suitable for this setting than gradient-based techniques, since the capabilities of the bots are more likely to be limited.

As we have seen in our experiments, using more significant swarms provides the capacity to generate adversarial examples with shorter $L_2$ distances. Therefore, by pooling resources together, a group of attackers can launch more powerful attacks that can generate adversarial examples that are increasingly harder to distinguish from their respective inputs. Such an attack could be scaled to accommodate inputs with higher dimensions and produce adversarial examples on models that are currently difficult to attack, such as those that are trained on 8K resolution images.

Additionally, as the bot nodes would likely be located in different geographical locations, monitoring incoming queries would be more complicated than if the attack is originating from a single host. Furthermore, decoy queries could be launched to confuse the target as to which entities are part of the attack. It could also be achieved by controlling the rate of queries being submitted by any single source, where queries are to be submitted in a specific

order, at different times, or in random bursts.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

In this dissertation, we present a black-box attack on ML models based on the evolutionary search algorithm Particle Swarm Optimization. Using this attack, we show that despite their high accuracy, classification models can be subverted by adversarial examples. We test attacks for both malware detection and image classification, and we show that in both domains, regardless of their inherent differences, models can still be fooled by specially crafted inputs.

For the malware classification model, we assume a behavioral-based target that utilizes API call sequences to classify software as malicious or benign. Attacks in this domain commonly employ an insertion technique that limits perturbations to feature additions to avoid disrupting the malware's original functionality. The API calls being inserted are null APIs, or no-ops, that have no functional effect, and would thus not change malware behavior. However, these methods can be detected using non-ML techniques by analyzing the inputs of these null APIs. For that reason, we chose to explore a more robust attack that generates adversarial examples by replacing API calls with other functionally-equivalent APIs.

To launch our attack against the API-based malware detection models, we first employ unsupervised learning techniques to find API call mappings that can be used in the attack. We apply these techniques on both API documentation and code, and in doing so, we were able to extract over 300 1-to-1 API mappings on the Windows platform. Using these mappings, we

generated adversarial examples from 28% of the malware samples in the our test set, outperforming a white-box gradient-based attack that share the same constraints.

In addition to attacking malware detection models, we also show in this dissertation the success of our attack against image classification models. Malware and images are inherently different, nonetheless, our attack successfully generated adversarial examples in both domains. To attack image classification models, we use query-reduction techniques such as the exploitation of the spacial relationship between pixels, minimizing redundant changes between particles, and using the edge of the $L_\infty$ ball when perturbing inputs. With these techniques, and by using multiple benchmarks, we show that AdversarialPSO performs comparably to the state-of-the-art and is able to generate adversarial examples with a limited query budget.

## 7.2 Future Work

The black-box attack we propose in this dissertation can be enhanced to become more powerful and more robust against defenses. The avenues we plan to explore to advance this attack are as following:

- This attack against the malware detection model can be further enhanced by extracting additional mappings. As our current attack only uses 1-to-1 mappings to perturb malware, a natural next step is to find 1-to-n, n-to-1, and n-to-n mappings that expands the search space for adversarial examples. These mappings are single API calls that can replace multiple consecutive calls (1-to-n), multiple calls that can be replaced by a single call (n-to-1), or multiple API calls that can be replaced by another sequence of API calls that perform the same functionally (n-to-n). Although finding such mappings would certainly enhance the efficacy of the attack, we leave this exploration for future work.

- In this dissertation, we focus on dynamic-based malware detection models that are trained on software behavior. Another type of malware detection models are ones that are trained on static-based features. The attack we propose can also be used on these models by replacing low-level assembly instructions found in malware binaries with equivalent

instructions. This type of manipulation is already exploited by metamorphic engines that modify malware samples to evade signature-based detection. However, these engines modify malware randomly and are not sufficient on their own to evade the detection of ML models. Our AdversarialPSO attack can be used to guide the manipulations made by the metamorphic engines to evade, not only signature-based detectors, but also ML-based models. We plan to explore this avenue further in future work.

- For the image classification attack, the main benefit we provide is the controllable trade-off between the number of queries and the quality of the adversarial examples. By using larger swarms, higher quality adversarial examples can be generated at the expense of more queries being submitted to the model. This property can be exploited to the extreme by launching a large-scale distributed attack from multiple remote origins, similar to how botnets operate. In launching a massive distributed attack, high-quality adversarial examples can be generated with a smaller detectable footprint, as the attack is launched from multiple different sources. It also allows the attack to be scaled to high-dimensional targets, such as models trained on 4K or 8K image, as these large inputs can be divided amongst all participants of the distributed attack. We leave the exploration of this large-scale distributed attack to future work.

- For both the malware and image classification attacks, other variants of PSO can be explored for further improvements. For example, an extension to PSO is Multi-Swarm Optimization (MSO), which uses multiple sub-swarms instead of a single swarm. Using MSO with different starting points could help cover more areas of the search space and eventually find better adversarial examples. Furthermore, another avenue is to utilize multiple sub-swarms that allow the use of different fitness functions or different swarm configurations. The sub-swarms could apply different distance penalties, mutate particles more or mutate them less, have different step sizes or have different exploitation/exploration/inertia weights. These different variants of PSO could provide added benefits and enhance the overall performance of the attack. We leave this exploration however, for future work.

# Bibliography


[1] Moustafa Alzantot, Bharathan Balaji, and Mani Srivastava. Did you hear that? adversarial examples against automatic speech recognition. In *Proceeedings of the 31st Conference on Neural Information Processing Systems*, NIPS 2017, 2017.

[2] Moustafa Alzantot, Yash Sharma, Supriyo Chakraborty, and Mani B. Srivastava. Genattack: Practical black-box attacks with gradient-free optimization. *CoRR*, abs/1805.11090, 2018.

[3] Blake Anderson, Curtis Storlie, and Terran Lane. Improving malware classification: bridging the static/dynamic gap. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, pages 3 – 14, 2012.

[4] Daniel Arp, Michael Spreitzenbarth, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of The 2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[5] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. The security of machine learning. *Machine Learning*, 81(2):121 – 148, 2010.

[6] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Practical black-box attacks on deep neural networks using efficient query mechanisms. In Martial Hebert, Vittorio Ferrari, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018 - 15th European Conference, 2018, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 158 – 174. Springer-Verlag, 1 2018.

[7] Daniel Bilar. Opcodes as predictor for malware. *Int. J. Electron. Secur. Digit. Forensic*, 1(2):156 – 168, 2007.

[8] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993 —- 1022, 2003.

[9] Nghi D. Q. Bui. Towards zero knowledge learning for cross language api mappings. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, pages 123 — 125. IEEE Press, 2019.

[10] Nghi D. Q. Bui and Lingxiao Jiang. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, pages 33 — 36, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39 – 57, 2017.

[12] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. *CoRR*, abs/1708.03999v2, 2017.

[13] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, PP(99), 2017.

[14] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303 – 317, San Diego, CA, 2014. USENIX Association.

[15] Susana C. Esquivel and Carlos A. Coello. On the use of particle swarm optimization with multimodal functions. In *The 2003 Congress on Evo-*

*lutionary Computation, 2003. CEC '03.*, volume 2, pages 1130 – 1136, 2003.

[16] Zwe-Lee Gaing. Particle swarm optimization to solving the economic dispatch considering the generator constraints. *IEEE Transactions on Power Systems*, 18(3):1187 – 1195, 2003.

[17] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[18] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *CoRR*, abs/1606.04435, 2016.

[19] Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples. *CoRR*, abs/1412.5068, 2014.

[20] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deepam: Migrate apis with multi-modal sequence to sequence learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 3675 — 3681. AAAI Press, 2017.

[21] Chuan Guo, Jacob R. Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Q. Weinberger. Simple black-box adversarial attacks. *CoRR*, abs/1905.07121, 2019.

[22] William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li. Dl4md: A deep learning framework for intelligent malware detection. In *Proceedings of the 12th International Conference on Data Mining*, pages 61 – 67, 2016.

[23] Matthew Hoffman, Francis R. Bach, and David M. Blei. Online learning for latent dirichlet allocation. In *Advances in Neural Information Processing Systems 23*, pages 856 – 864. Curran Associates, Inc., 2010.

[24] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on GAN. *CoRR*, abs/1702.05983, 2017.

[25] Weiwei Hu and Ying Tan. Black-box attacks against rnn based malware detection algorithms. In *AAAI Workshops*, 2018.

[26] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. Mutantx-s: Scalable malware clustering based on static features. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 187 – 198, San Jose, CA, 2013. USENIX.

[27] Alex Huang, Abdullah Al-Dujaili, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. *CoRR*, abs/1801.02950, 2018.

[28] Wenyi Huang and Jack W. Stokes. Mtnet: A multi-task neural network for dynamic malware classification. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 399 – 418. Springer-Verlag New York, Inc., 2016.

[29] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. *CoRR*, abs/1804.08598, 2018.

[30] Andrew Ilyas, Logan Engstrom, and Aleksander Madry. Prior convictions: Black-box adversarial attacks with bandits and priors. *CoRR*, abs/1807.07978, 2018.

[31] Hesam Izakian, Behrouz Tork Ladani, Kamran Zamanifar, and Ajith Abraham. A novel particle swarm optimization approach for grid job scheduling. In Sushil K. Prasad, Susmi Routray, Reema Khurana, and Sartaj Sahni, editors, *Information Systems, Technology and Management*, pages 100 – 109, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[32] Ahmad Rezaee Jordehi and Jasronita Jasni. Particle swarm optimisation for discrete optimisation problems: a review. *Artificial Intelligence Review*, 43(2):243 – 258, Feb 2015.

[33] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942 – 1948, 1995.

[34] James Kennedy and Russell C. Eberhart. A discrete binary version of the particle swarm algorithm. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, volume 5, pages 4104–4108 vol.5, Oct 1997.

[35] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. A novel approach to detect malware based on api call sequence analysis. *International Journal of Distributed Sensor Networks*, 11(6):659101, 2015.

[36] Bob Klein and Ryan Peters. Defeating machine learning - what your security vendor is not telling you. In *Proceedings of Black Hat 2015 USA*, 2015.

[37] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In Byeong Ho Kang and Quan Bai, editors, *AI 2016: Advances in Artificial Intelligence*, pages 137 – 149, Cham, 2016. Springer International Publishing.

[38] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016.

[39] Kaspersky Lab. It threat evolution in q2 2019. Technical report, Kaspersky Lab, 2019.

[40] Kaspresky Lab. Machine learning for malware detection. Whitepaper, Kaspresky for Business, 2019.

[41] Elena C. Laskari, Konstantinos E. Parsopoulos, and Michael N. Vrahatis. Particle swarm optimization for integer programming. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, volume 2, pages 1582 – 1587, May 2002.

[42] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[43] Jiwei Li, Minh-Thang Luong, and Dan Jurafsky. A hierarchical neural autoencoder for paragraphs and documents. *CoRR*, abs/1506.01057, 2015.

[44] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129 –137, 1982.

[45] Yonghe Lu, Minghui Liang, Zeyuan Ye, and Lichao Cao. Improved particle swarm optimization algorithm and its application in text feature selection. *Applied Soft Computing*, 35:629 – 636, 2015.

[46] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *International Conference on Learning Representations*, 2018.

[47] Zane Markel and Michael Bilzor. Building a machine learning classifier for malware detection. *2014 Second Workshop on Anti-malware Testing Research (WATeR)*, 2014.

[48] Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. *CoRR*, abs/1705.09064, 2017.

[49] Microsoft. Use next-gen technologies in windows defender antivirus through cloud-delivered protection, 2019.

[50] Najmeh Miramirkhani, Mahathi Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, 2017.

[51] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. Amal: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52:251 – 266, 2015.

[52] Seungyong Moon, Gaon An, and Hyun Oh Song. Parsimonious black-box adversarial attacks via efficient combinatorial optimization. *CoRR*, abs/1905.06635, 2019.

[53] Shraddha More and Pranit Gaikwad. Trust-based voting method for efficient malware detection. In *7th International Conference on Communication, Computing and Virtualization*, pages 657 – 667. Elsevier, 2016.

[54] Rayan Mosli, Rui Li, Bo Yuan, and Yin Pan. Automated malware detection using artifacts in forensic memory images. In *Proceedings of IEEE Symposium on Technologies for Homeland Security (HST)*, 2016.

[55] Rayan Mosli, Rui Li, Bo Yuan, and Yin Pan. Getting a handle on malware detection: A behavioral-based approach. In *Proceedings of 13th Annual IFIP WG 11.9 International Conference on Digital Forensics*. Springer, 2017.

[56] Saeed Nari and Ali Ghorbani. Automated malware classification based on network behavior. In *Proceedings of International Conference on Networking and Communications (ICNC)*, pages 642 – 647, 2013.

[57] Hiran Nath and Babu Mehtre. Static malware analysis using machine learning methods. *Communications in Computer and Information Science Recent Trends in Computer Networks and Distributed Systems Security*, 420, 2014.

[58] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D. Joseph, Benjamin I. P. Rubinstein, Udam Saini, Charles Sutton, J. D. Tygar, and Kai Xia. Exploiting machine learning to subvert your spam filter. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08, pages 7:1 – 7:9, 2008.

[59] Gary Pampara, Nelis Franken, and Andries P. Engelbrecht. Combining particle swarm optimisation with angle modulation to solve binary problems. In *2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 89 – 96, Sep. 2005.

[60] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 506–519, New York, NY, USA, 2017. ACM.

[61] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 372 – 387, 11 2016.

[62] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. *CoRR*, abs/1511.04508, 2015.

[63] Nicolas Papernot, Patrick Drew McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. *CoRR*, abs/1511.04508, 2015.

[64] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. 2019.

[65] Radu Pirscoveanu, Steven Hansen, and Alexandre Czech. Analysis of malware behavior: Type classification using machine learning. In *Proceedings of the 2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, 2015.

[66] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole exe. In *The Workshops of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[67] Matilda Rhode, Pete Burnap, and Kevin Jones. Early stage malware prediction using recurrent neural networks. *CoRR*, abs/1708.03513, 2017.

[68] Ishai Rosenberg, Shabtai Asaf, Yuval Elovici, and Lior Rokach. Query-efficient black-box attack against sequence-based malware classifiers. *CoRR*, abs/1804.08778, 2018.

[69] Christian Rossow, Christian Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[70] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987.

[71] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017.

[72] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, pages 64–82, 2013.

[73] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015.

[74] Yuhui Shi and Russell C. Eberhart. Empirical study of particle swarm optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 3, pages 1945 – 1950, 02 1999.

[75] Nedim Srndic and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE Symposium on Security and Privacy*, pages 197 – 211, 2014.

[76] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *CoRR*, abs/1710.08864, 2017.

[77] Symantec. Machine learning: New frontiers in advanced threat detection, 2019.

[78] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199v4, 2014.

[79] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013.

[80] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey on the usage of machine learning techniques for malware analysis. *CoRR*, abs/1710.08189, 2017.

[81] Phani Vadrevu and Roberto Perdisci. Maxs: Scaling malware execution with sequential multi-hypothesis testing. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 771 — 782, New York, NY, USA, 2016. Association for Computing Machinery.

[82] Kalyan Veeramachaneni, Lisa Osadciw, and Ganapathi Kamath. Probabilistically driven particle swarms for optimization of multi valued discrete problems : Design and analysis. In *2007 IEEE Swarm Intelligence Symposium*, pages 141 – 149, April 2007.

[83] Wei Yang, Deguang Kong, Tao Xie, and Carl A. Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 288 – 302, 2017.